



LinuxScope-JTD™ Installation Guide

Version 4.0.0

Platform Support

LinuxScope-JTD v4.0.0 has been tested on the following platforms:

Linux Ubuntu 10.04 and 11.10 (others will probably work)

Windows XP (Other 32-bit NT-based systems such as Windows NT 4, Windows 2000, and other Windows XP versions will probably work)

Hardware Requirements

1GHz CPU recommended, 1GB of RAM

800MB free disk space (depending on installation method, less may suffice)

Prerequisites

BDI Configuration

It is assumed that the BDI has already been configured (given an IP address and loaded with firmware appropriate for the target). LinuxScope-JTD does not configure the BDI, nor does it come bundled with BDI firmware. See the BDI documentation for information about configuring the probe.

TFTP Server

A TFTP server is also required; this is used to provide files to the BDI. LinuxScope-JTD does not currently come with a TFTP server. Linux has a built in server, while a good freeware TFTP server for Windows can be found at <http://tftpd32.jounin.net>. See the BDI documentation for information about setting up a TFTP server.

Java VM

A stable Java VM is required. The latest Sun 1.6.x VM will work, others (especially GNU “gij”) may be insufficient. The installation CD contains installation packages of Sun's Java Runtime Environment for supported platforms. Or, to download and install an appropriate Java VM, please visit:

<http://java.sun.com/javase/downloads/index.jsp>

A full list of operating systems and JVMs tested with Eclipse can be found here:

http://www.eclipse.org/projects/project-plan.php?projectid=eclipse#target_environments

GCC

For the Eclipse CDT to work properly, development tools such as 'gcc' must be installed and included in the path. See the platform-specific notes. These are not needed for debugging, but are essential for building binaries under Eclipse. If Eclipse will not be used to build images, gcc is not required.

GDB

GDB forms the foundation for all debugging activity in LinuxScope-JTD, so a good version is required. For optimal performance, a version must be used that contains extensions added by the LinuxScope development team. (These extensions fix some not-yet-implemented MI commands and will be submitted to the DMI working group for inclusion in future GDB versions). LinuxScope-JTD comes with pre-built GDB 6.8 binaries for supported platforms and targets, which can be directly copied to the host machine. The source code for this GDB release, along with the extensions, is provided on the installation disc so that builds for additional targets and platforms can be made.

Installation Overview

LinuxScope-JTD can be installed in two ways, as a standalone product or a plug-in to an existing Eclipse-based product. Standalone installation is the simplest method. There are installers/scripts provided to simplify the standalone install process.

Note: When you launch Eclipse, it asks for a workspace path. This is the location where all projects, settings, code, data, etc. are stored. Do not select a path with a space in it, as this can cause failures with external tools, such as GDB.

Standalone Installation

NOTE: A full eclipse platform and LinuxScope product will automatically be installed when doing the Standalone Installation. There is no need to perform any Plugin Installation.

Windows Installation

Simply run the "LinuxScopeJTD_Installer.exe" in the windows folder on the CD and follow the prompts.

Linux Installation

Choose the appropriate binary directory (linux_x86-bin or linux_x86_64-bin), and execute the install.sh script. On the first run, the End User License Agreement will be presented. To accept the license and install the program, run install.sh again with the word 'accept' as an argument. Also provide the installation path, for example:

```
sh install.sh accept /opt/LinuxScopeJTD
```

Manual Installation

To install LinuxScope-JTD as a standalone application, simply extract the archive appropriate for the platform. The archive can be extracted to any desired location. The archive contains a branded version of Eclipse, the Eclipse CDT, and LinuxScope-JTD. To launch LinuxScope-JTD, execute the 'Linuxscope' binary in the 'linuxscope' folder. Adding a GUI shortcut is platform-specific and is not done automatically.

When launching Eclipse, it is a good idea to specify the Java VM to use, especially if more than one is installed. This can be done by specifying command line arguments to eclipse, for example:
eclipse -vm c:\jdk1.6.12\jre\bin\java

Plug-in Installation

Using LinuxScope-JTD as a plug-in requires:

Eclipse Platform (4.2.0 is required)

Eclipse CDT (8.1.0 is required)

Installing the Eclipse Platform

Eclipse 4.2.0 is required. This can be downloaded from:

<http://www.eclipse.org/downloads/packages/release/ganymede/r>

“Eclipse IDE for C++ Developers” is recommended as it comes bundles with CDT 8.1; Other platforms can also be used and latter updated with the CDT plugin.

Once downloaded, the archive should be extracted to the location where Eclipse should be installed.

When launching eclipse, it is a good idea to specify the Java VM to use, especially if more than one is installed. This can be done by specifying command line arguments to eclipse, for example:

```
eclipse -vm c:\jdk1.6.0\jre\bin\java
```

For more information, see:

http://www.eclipse.org/eclipse/development/readme_eclipse_3.4.1.html

Shortcuts to launch eclipse are platform-dependent and are not automatically added.

Note: When you launch Eclipse, it asks for a workspace path. This is the location where all projects, settings, code, data, etc. are stored. Do not select a path with a space in it, as this can cause failures with external tools, such as GDB.

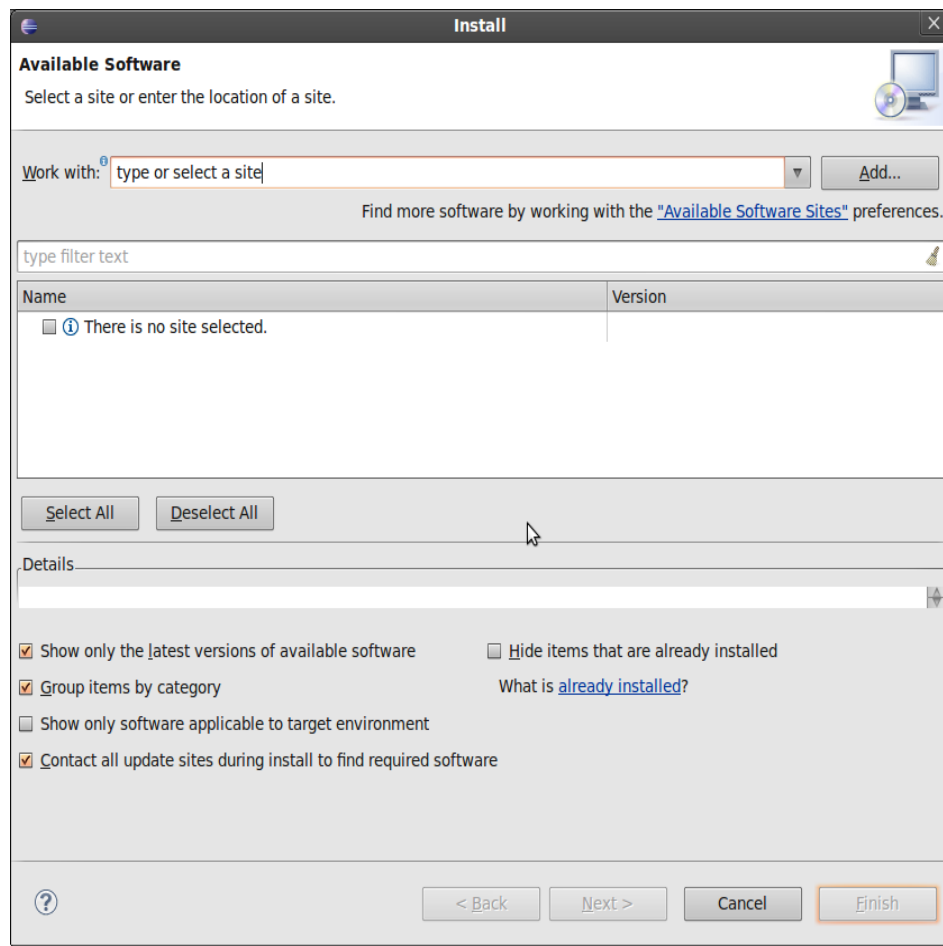
To facilitate collection of logging information during beta testing, debug logging should be enabled. Add the following line to the '.options' file in the 'eclipse' directory (if '.options' does not exist, create it):

```
org.eclipse.cdt.debug.mi.core/debug=true
```

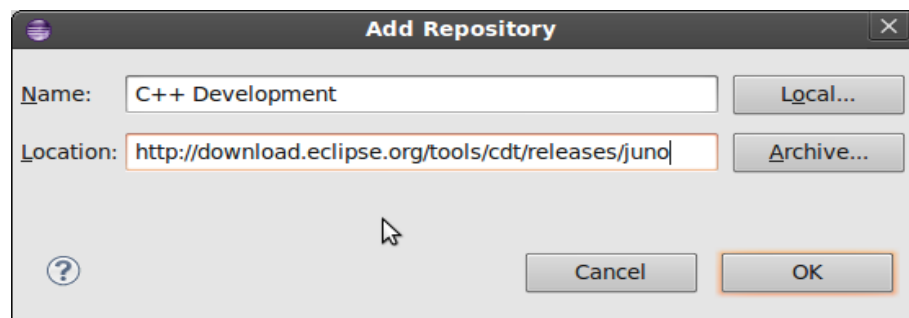
Also, specify the '-debug' argument on the command line when running Eclipse.

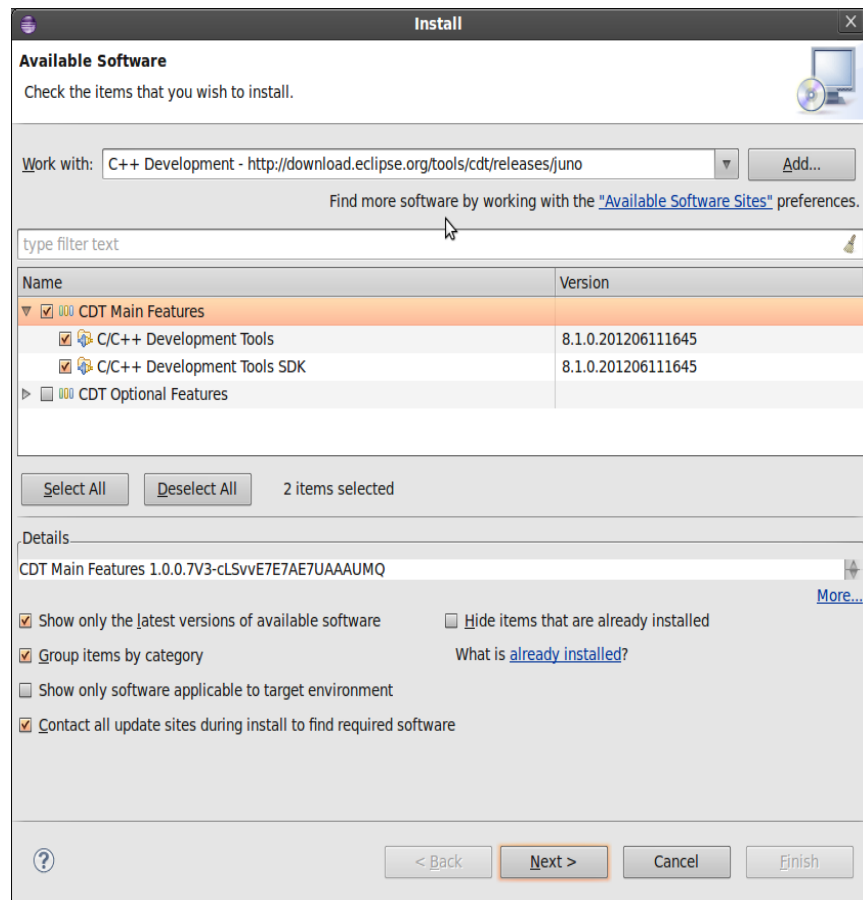
Installing the CDT

The C/C++ Development Tools (CDT) is installed from within Eclipse itself. Run Eclipse and open the “Help” menu. Select “Install New Software...”. The following wizard should appear:

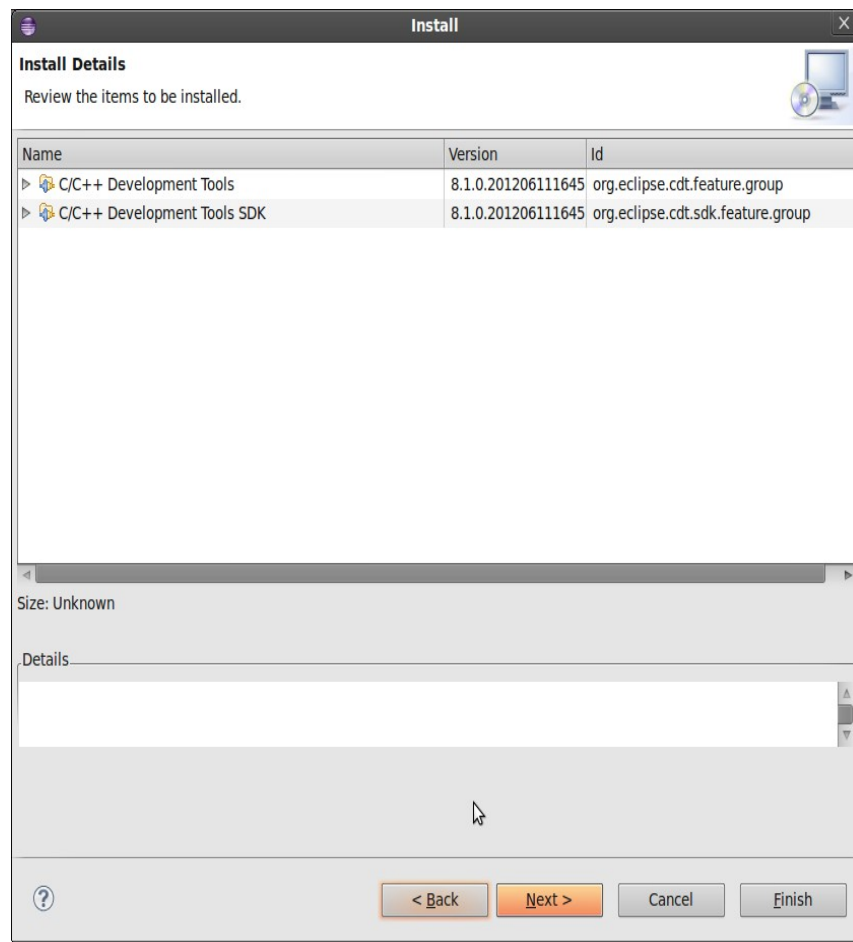


Click the “Add” button, enter the information below, and click OK.

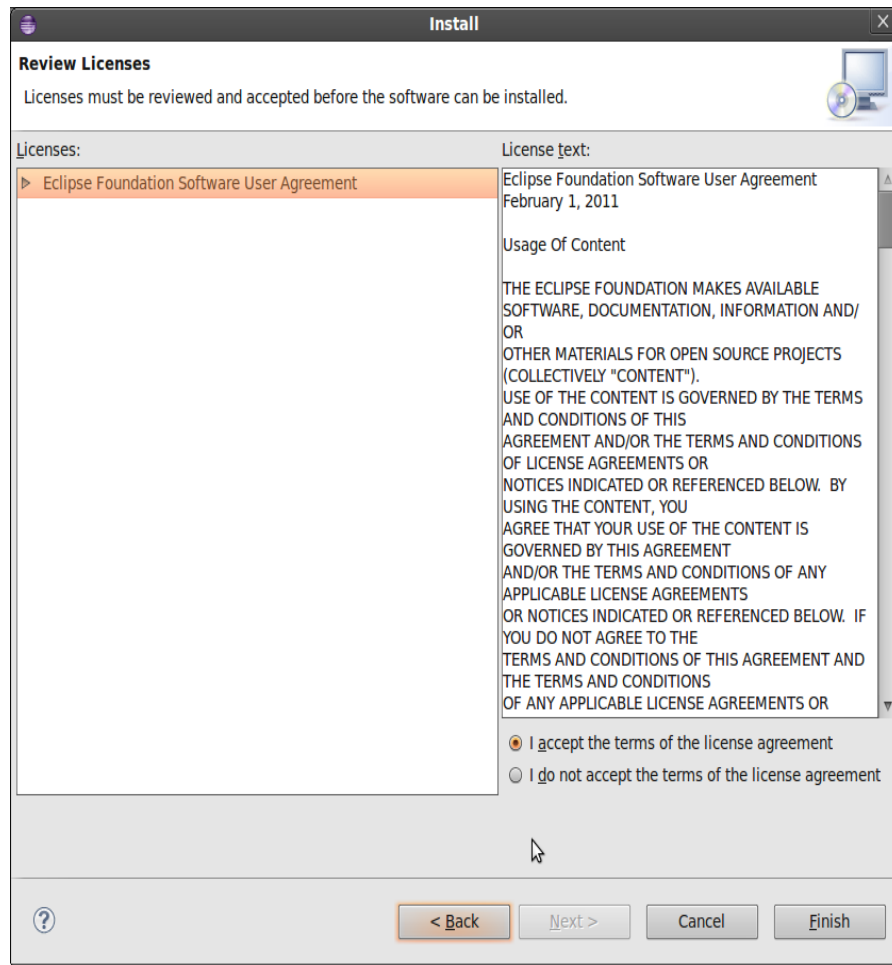




Check the “CDT Main Features” box and click “Next”. Click “Next” again.



Accept the License agreement and click “Finish”.



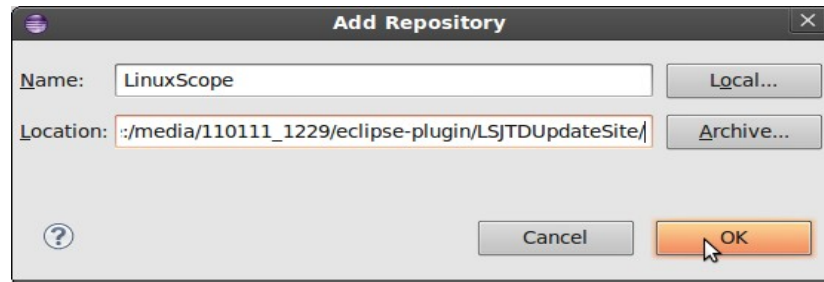
A progress bar will show the status of the Install, once finished CDT should then be installed. Select “Yes” and allow Eclipse to restart the workspace once CDT is installed.

Note: If you wish to compile programs in Eclipse CDT, you must install GCC. CDT does not tend to give explicit errors when GCC is not properly installed, so if it is “behaving funny” and doesn't seem to offer the expected array of actions, this is a good thing to check.

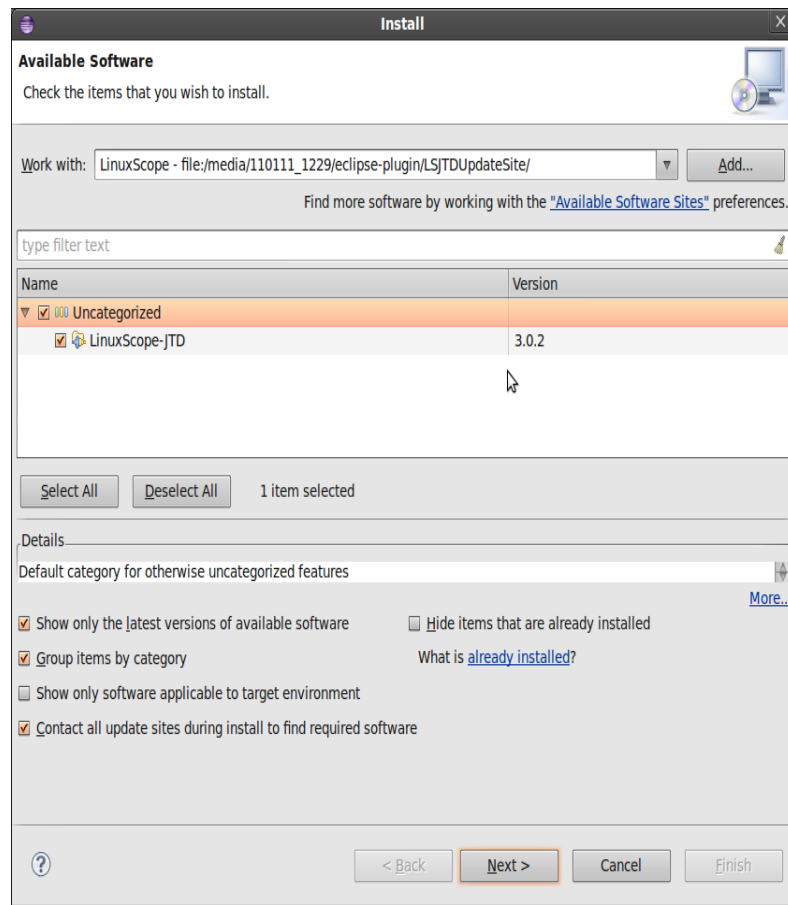
Installing LinuxScope-JTD Plug-In

Installing the LinuxScope-JTD Eclipse plug-in is similar to installing the CDT:

1. Open “Help->Install New Software...”
2. Create a new local update site that points to the “eclipse-plugin/LSJTDUpdateSite” folder from the installation media:



3. Select “LinuxScope-JTD 4.0.0” and “Next”:



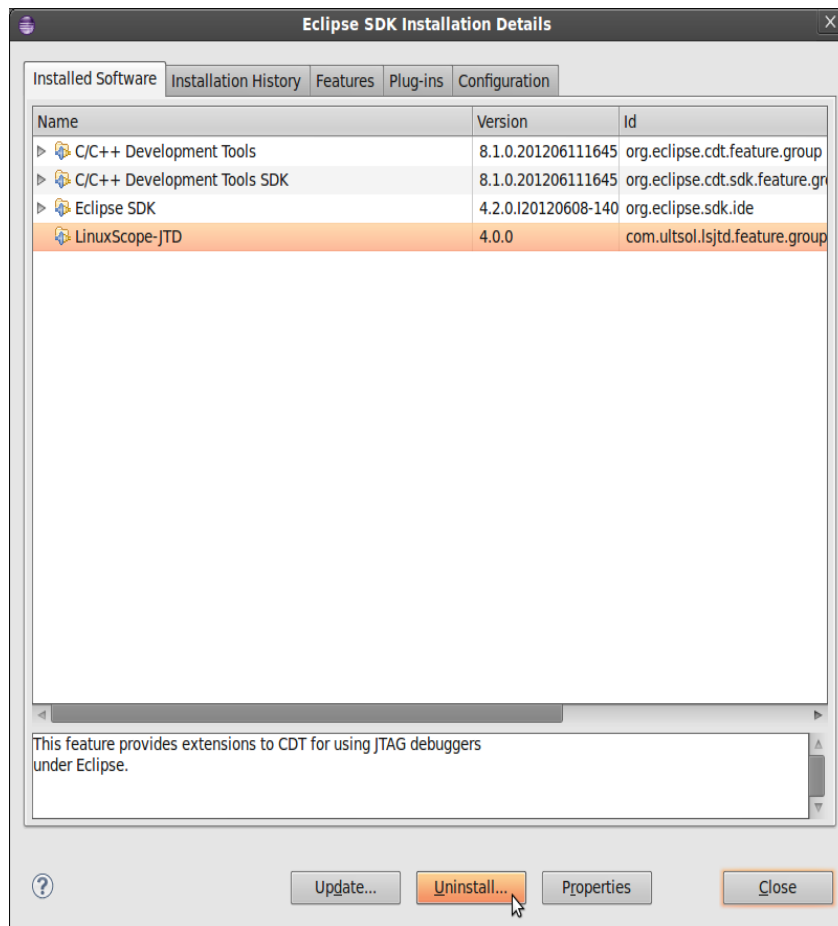
4. Review and accept the license terms
5. Select “Finish”
6. Select “Install All” if presented with “Feature Verification” dialogs
7. Select “Yes” and allow Eclipse to restart the workspace with the new plug-ins

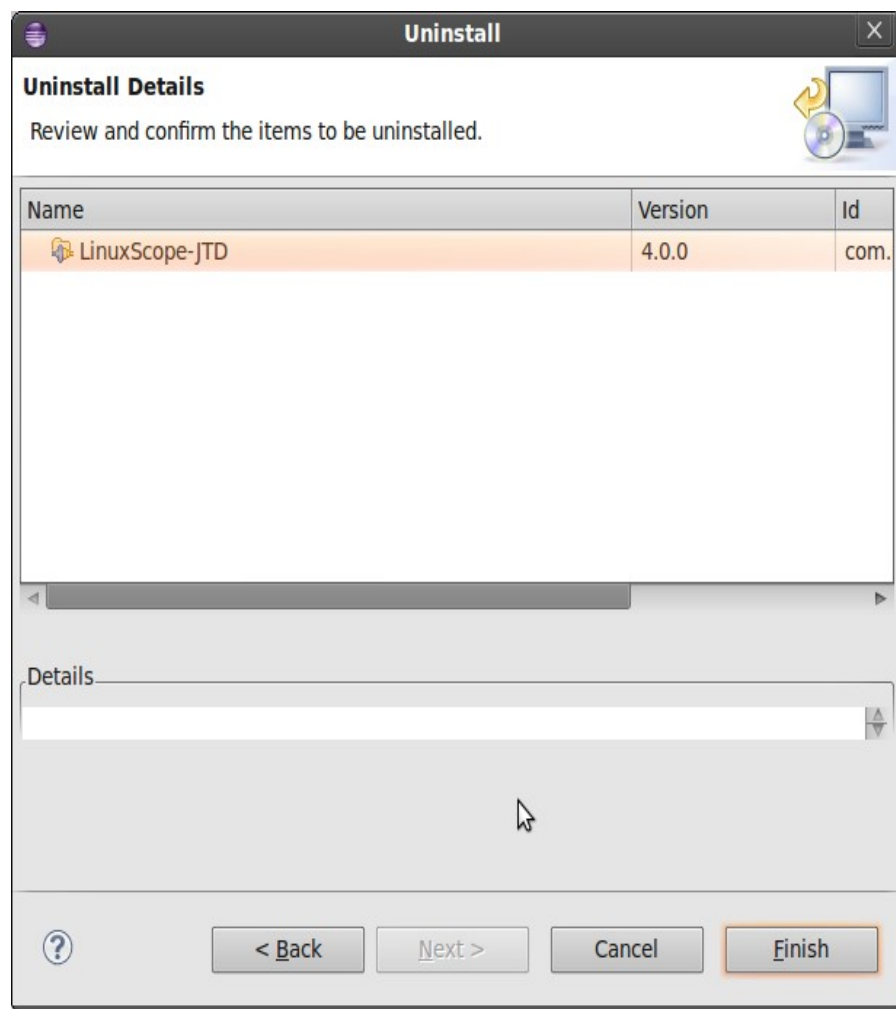
Uninstalling Plug-ins/Features

To uninstall plug-ins (e.g. Prior to installing newer versions), select “Help->About Eclipse SDK”.



Click “Installation Details”. On the “Installed Software” tab, select LinuxScope-JTD and click “Uninstall”.





Review and click “Finish”.

Platform-specific Installation Notes

Windows Platforms

Tools such as GCC and GDB are typically run under Cygwin, which is a layer that provides Unix-like APIs on Windows. To get Cygwin, go to <http://www.cygwin.com> and run the setup.exe. You can also use MINGW and MSYS www.mingw.com. The GDB binaries provided with LinuxScope-JTD were built with mingw and msys.

Installation Verification/Troubleshooting

Review the following screens to make sure the information is reasonable:

1. “Help->About Eclipse”. Select “Configuration” tab: Make sure the proper VM is being used. This screen also provides a way to capture this information to the clipboard; this allows the configuration to be submitted in support requests.
2. “Help->About Eclipse”. Select “Plugins” tab: Make sure the “Ultimate Solutions” `com.ultsol.lsjtd.ui` plug-in is listed.

Building GDB

If the bundled GDB versions are insufficient for a given target, additional GDB builds can be made using the following procedure:

1. Make sure GCC and GNU make are installed; these are necessary to compile GDB.
2. Extract the contents of gdb6.8-src.tar.gz (located in the gdb-src directory on the installation CD) to a temporary working location, using a command such as 'tar -xzf gdb6.8-src.tar.gz'.
3. Change to the gdb-6.8 directory and configure GDB using a command similar to the following:

```
./configure --target=powerpc-linux-gnu --program-prefix=ppc-  
--disable-gdbtk --disable-gdbgui --without-x
```

Replace the argument to --target with something appropriate; in general this is

--target=<architecture>-<vendor>-<os>. There is no easy way to get a list, but reading bfd/config.bfd may provide some insights into what is accepted. Also, use a program prefix indicative of the target supported.

4. Run make to produce and install a binary:

```
make && make install
```

5. Clean up the intermediate files so that GDB can be configured and built again:

```
make distclean
```



LINUX SCOPE-JTD

JTAG TARGET DEBUGGER

LinuxScope-JTD™ User's Manual

Version 4.0.0

Table of Contents

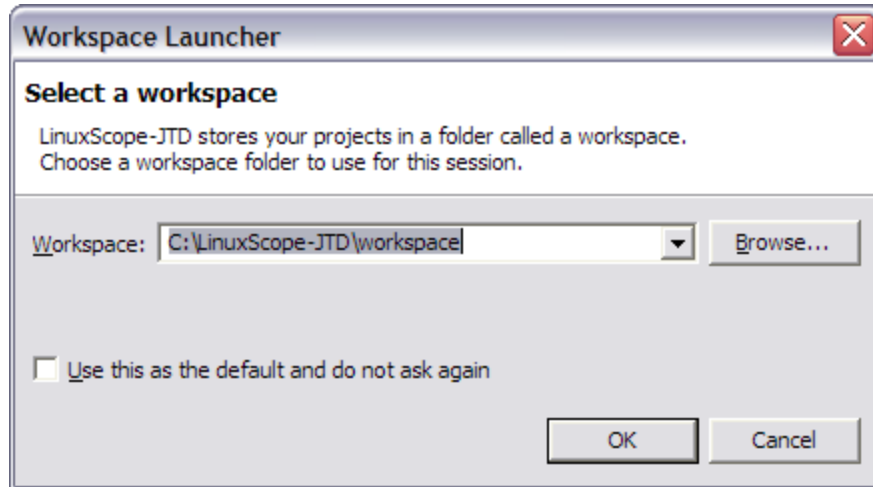
1. Quick Start.....	4
1.1 Welcome.....	4
1.2 LinuxScope-JTD cheat sheet.....	6
2 Using Eclipse and LinuxScope-JTD.....	14
2.1 Configuring LinuxScope-JTD and the JTAG Probe.....	14
2.2 Other Useful Preferences.....	16
2.2.1 Console.....	16
2.2.2 GDB Timeouts.....	17
2.2.3 Source Editor Preferences.....	18
2.2.4 Path Mapping.....	19
3 Working with LinuxScope-JTD.....	20
3.1 Launching Perspectives.....	20
3.2 Command Tool.....	21
3.3 Creating a Project.....	24
3.4 Launching Debug Sessions.....	27
3.5 Creating A Linux Build Project.....	32
4 Source Debugging.....	38
4.1 Connecting To Target.....	38
4.2 Setting Breakpoints.....	38
4.3 Setting Data Breakpoints.....	38
4.4 Finding Function and Variable Declarations.....	40
5 JTAG Debug Perspective Tour.....	43
5.1 JTD Registers View.....	44
5.2 JTD Modules View.....	45
5.2.1 Adding Symbol Files.....	46
5.3 Finding Breakpoint Symbols.....	48
5.4 Controlling Debugging with the Debug View.....	49
5.5 Console View.....	51
5.6 Scripting.....	52
5.7 Telnet View.....	58
5.8 Disassembly View.....	59
5.9 Breakpoints View.....	60
5.10 Variables, Expressions, and Memory Views.....	61
6 Flashing.....	62

6.1 NOR.....	62
6.2 NAND.....	63
7 General Tips.....	65

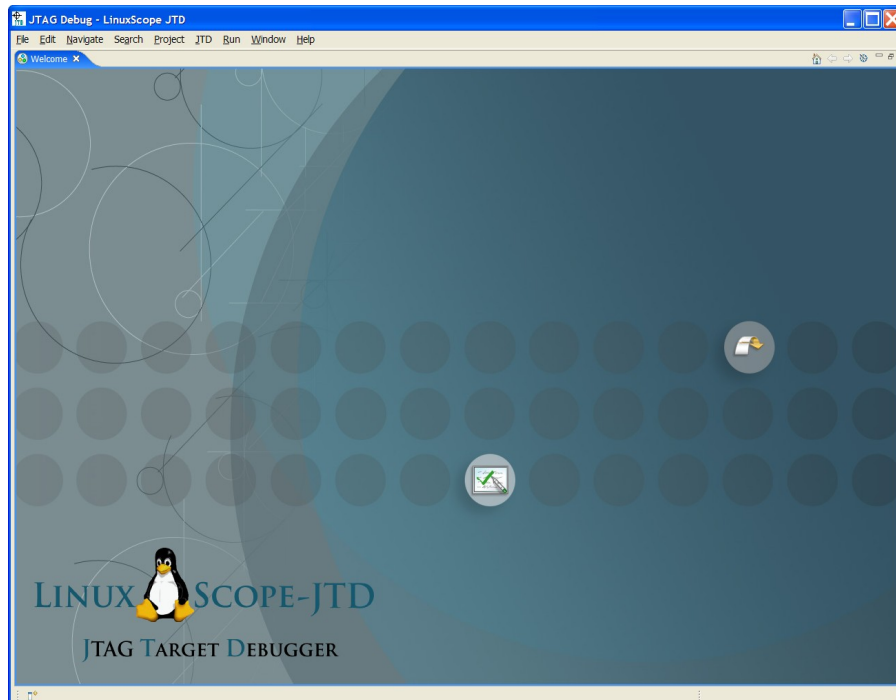
1. Quick Start

1.1 Welcome

When LinuxScope-JTD is run for the first time, the workspace must be selected. This is the folder where Eclipse will store all projects and settings. This path should not contain spaces.

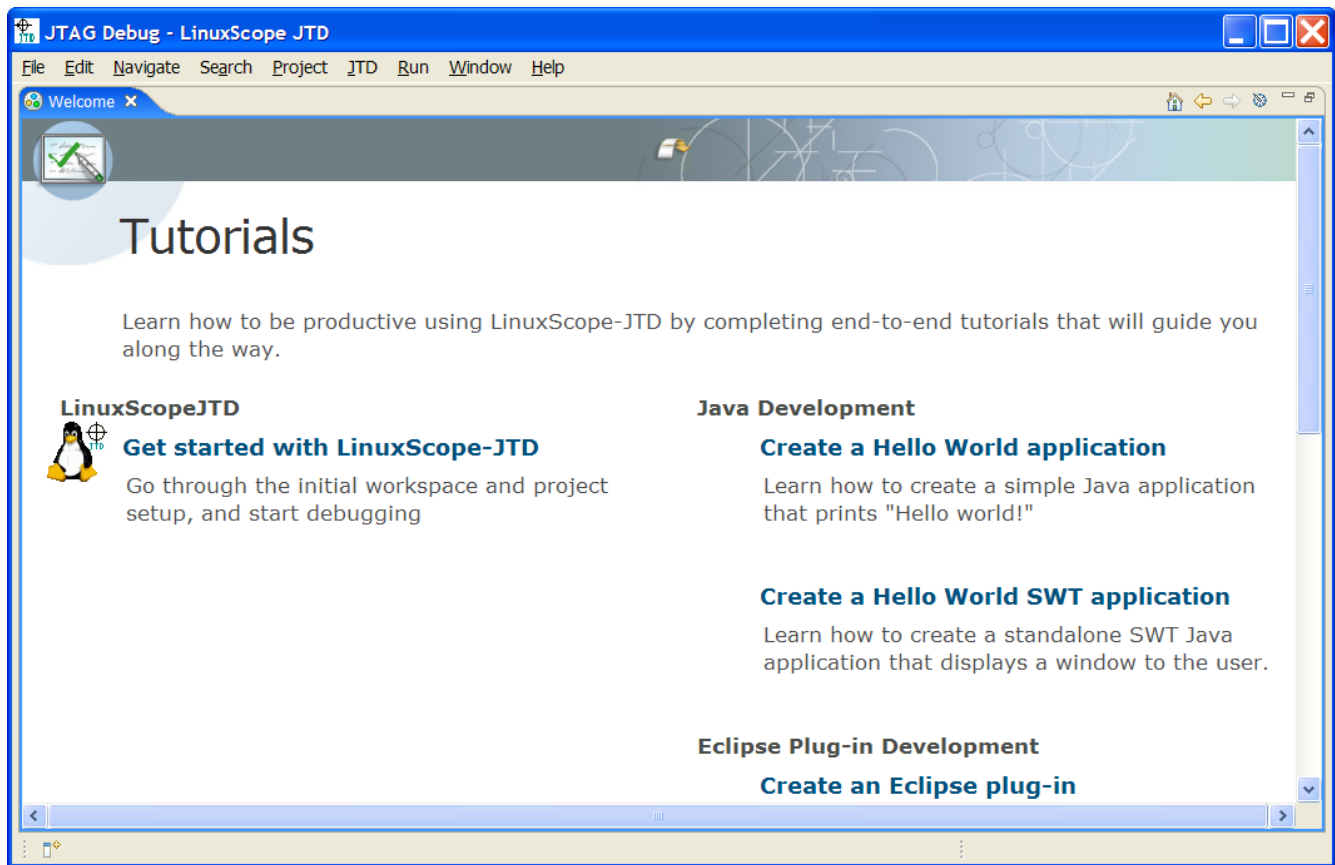


Once the workspace has been selected, the welcome screen should appear:



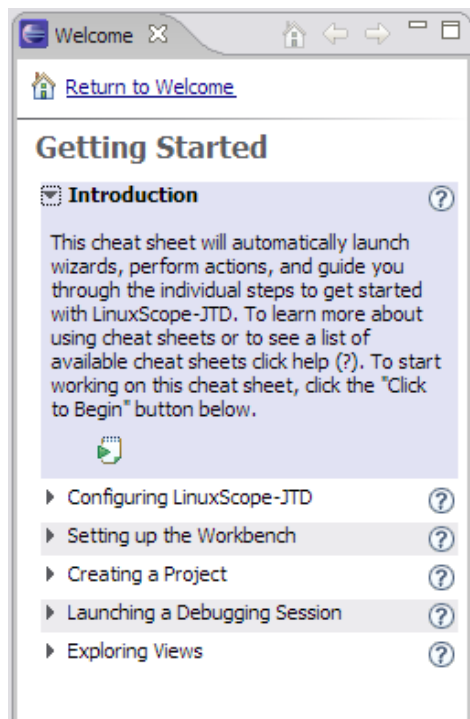
This screen can offer several items that may be useful to Eclipse beginners. The tutorials section (bottom center) contains a “Getting Started” cheat sheet to help you get started with LinuxScope-JTD.

To access the tutorial, click the “Tutorials” icon and then the LinuxScope-JTD entry.



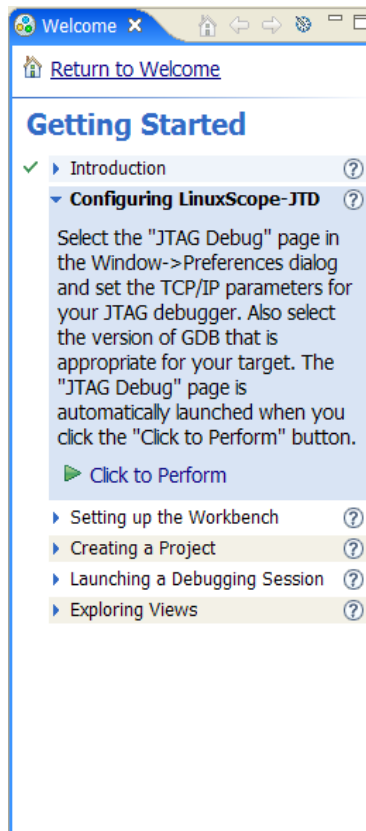
This brings up the LinuxScope-JTD cheat sheet.

1.2 LinuxScope-JTD cheat sheet

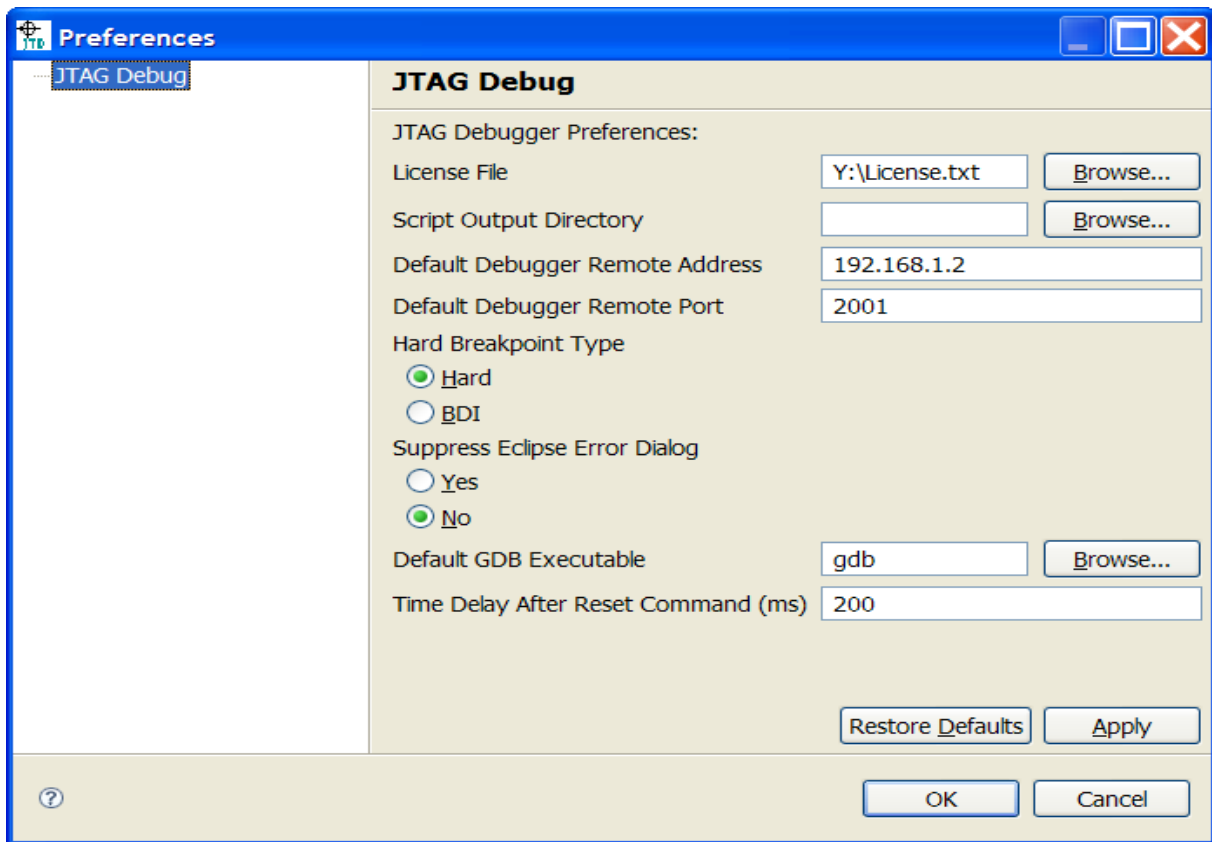


The dialogs and pages launched by the cheat sheet is described in the sections to follow.

The initial cheat sheet page is titled “Introduction”. Clicking the “Click to Begin” button launches the “Configuring LinuxScope-JTD” page.



Click to perform will launch a preferences dialog.



Use this dialog box to set the defaults the BDI will use every time it launches a new debug session. These default settings can be overridden when you setup a new debug session.

License File: Be sure to specify a valid License file. License files for LinuxScope can be obtained from Ultimate Solutions Inc., <http://www.ultsol.com/license.htm>, Phone(USA&Canada): (978) 455 3383.

Script Output Directory: Directory where all script output files will be saved.

Default Debugger Remote Address: IP address of the JTAG Probe you will be connecting to.

Default Debugger Remote Port: The port number where you will communicate with the JTAG Probe.

Hard Breakpoint Type: Chose to set a breakpoint directly on the BDI or through eclipse.

Suppress Eclipse Error Dialog: Stop error dialog boxes from popping up all the time.

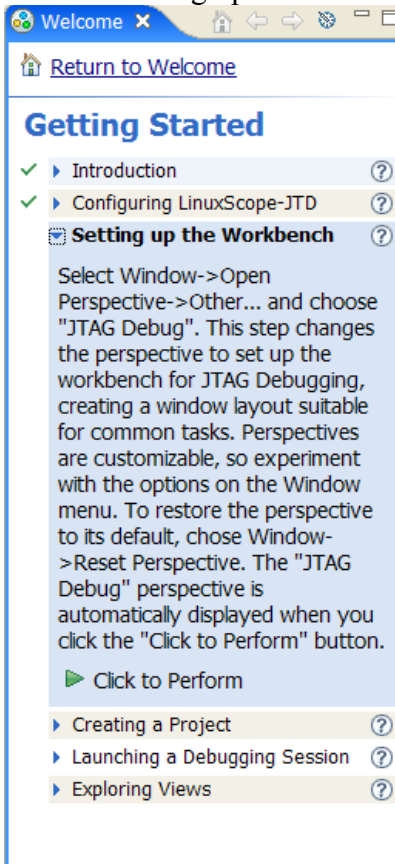
Default Breakpoint Type: Forces the JTAG Probe to use Hardware or Software Breakpoints.

Default GDB Executable: Several gdb executables are provided with LinuxScope in the /gdb folder, use one that most closely resembles your debug target.

Time Delay after reset Command: Increase this delay if your system takes a long time to reset.

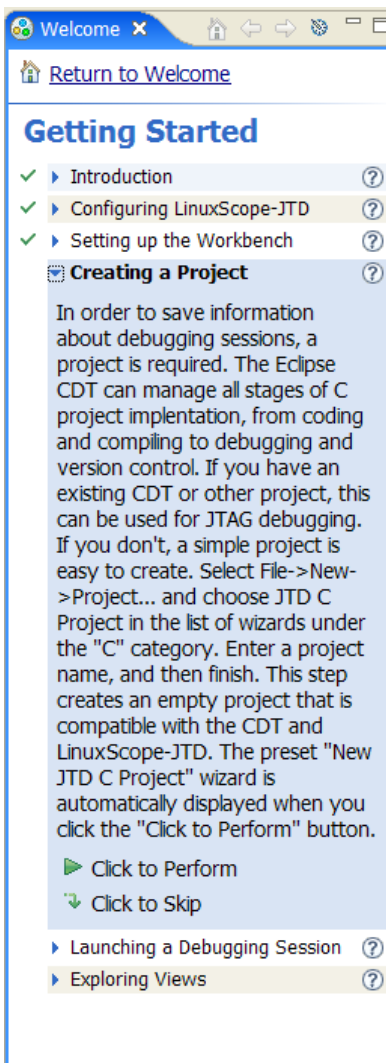
When finished click “apply” then “ok”.

The next cheat sheet page will launch titled “Setting up the workbench”

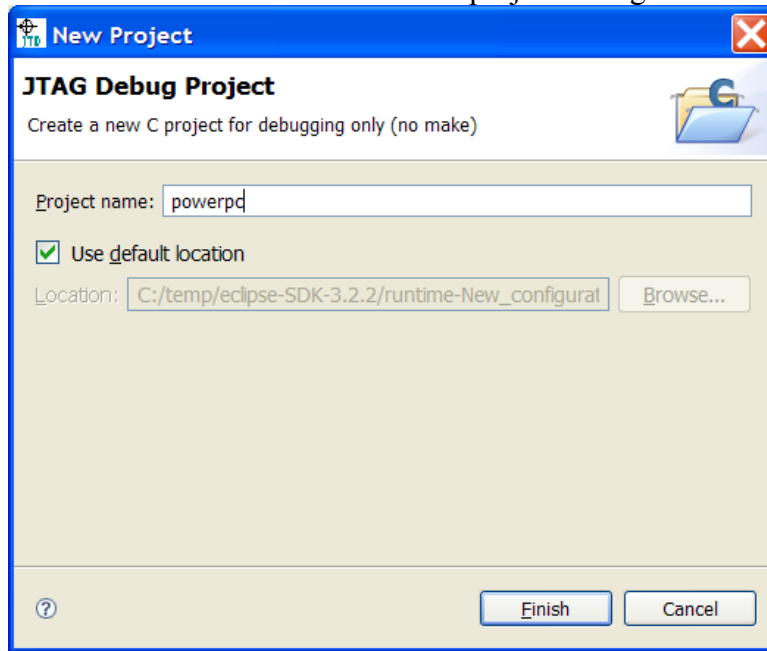


“Click to Perform” will launch the “JTAG_Debug” perspective view in Eclipse if it is not already open.

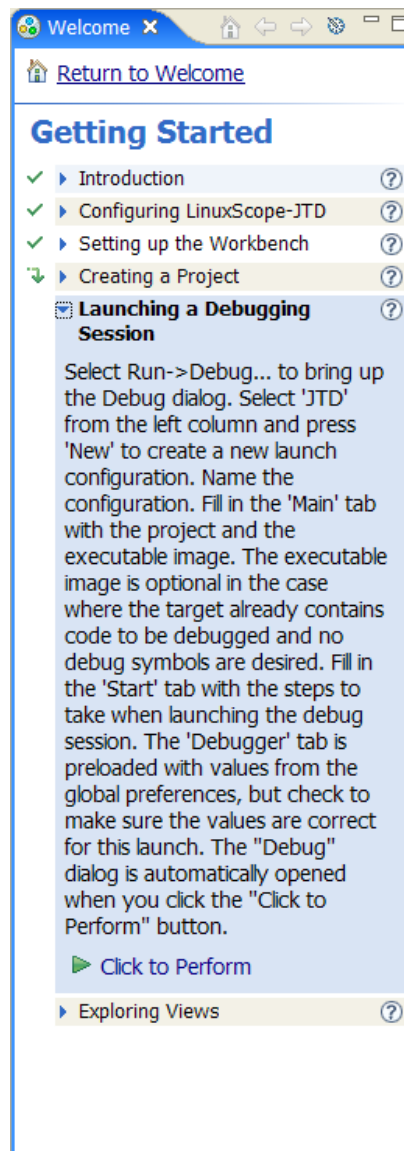
The cheat sheet will now display the next page titled “Creating a project”



Click on the “Click to Perform” button to launch the new project dialog.



Specify a new project name and click finish, or click cancel if a project to be used already exists. The cheat sheet will now display the “Launching a Debugging Session” page.

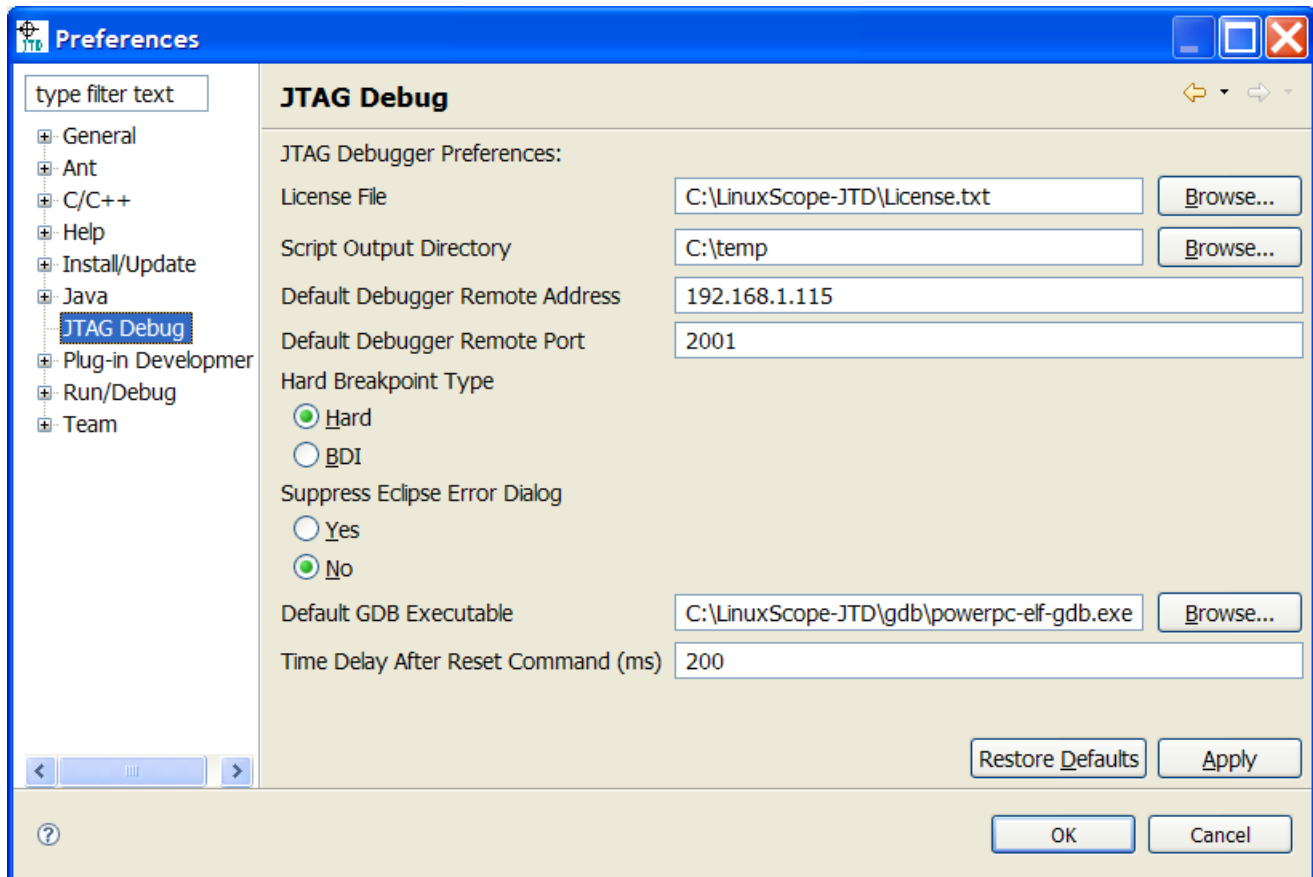


By clicking “Click to Perform” you will be taken to the launch debug session dialog. Refer to section 3.4, “Launching Debug Sessions”, of this manual.

2 Using Eclipse and LinuxScope-JTD

2.1 Configuring LinuxScope-JTD and the JTAG Probe

Global LinuxScope preferences can be set in the “Preferences” dialog. Open the preferences dialog by selecting “Window->Preferences...” from the main menu bar.



Select “JTAG Debug” from the tree to display the JTAG preferences page. The fields set default values for probe parameters; these values can be overridden on a per-target basis (see the section “Launching Debug Sessions” for details).

License File: This is the path to the license file supplied by Ultimate Solutions or a franchised distributor. The license is a text file that can be saved anywhere in the file system.

Script Output Directory: This is the directory where the captured output of executed scripts will be stored.

Default Debugger Remote Address: This is the IP address of the JTAG probe. This is a default that can be overridden in each launch configuration.

Default Debugger Remote Port: This is the TCP port number of the JTAG probe. This is a default that can be overridden in each launch configuration. This value should not be modified in most cases.

Hard Breakpoint Type: By default this option is set to “Hard”. In this mode you can set hardware breakpoints from the GUI and Linuxscope-JTD will keep track of your breakpoints. If for some reason you encounter errors setting or hitting hardware breakpoints you can chose the alternate mode “BDI”. In this mode the breakpoints are set directly on the JTAG probe and Linuxscope-JTD will clear any

previous hardware breakpoint before setting the next one. LinuxScope-JTD cannot keep track of breakpoints when “BDI” is selected. Breakpoints listed under the breakpoints view are meaningless.

Suppress Eclipse Error Dialog: Default is no. Use this option if you get error dialog boxes consistently during debugging.

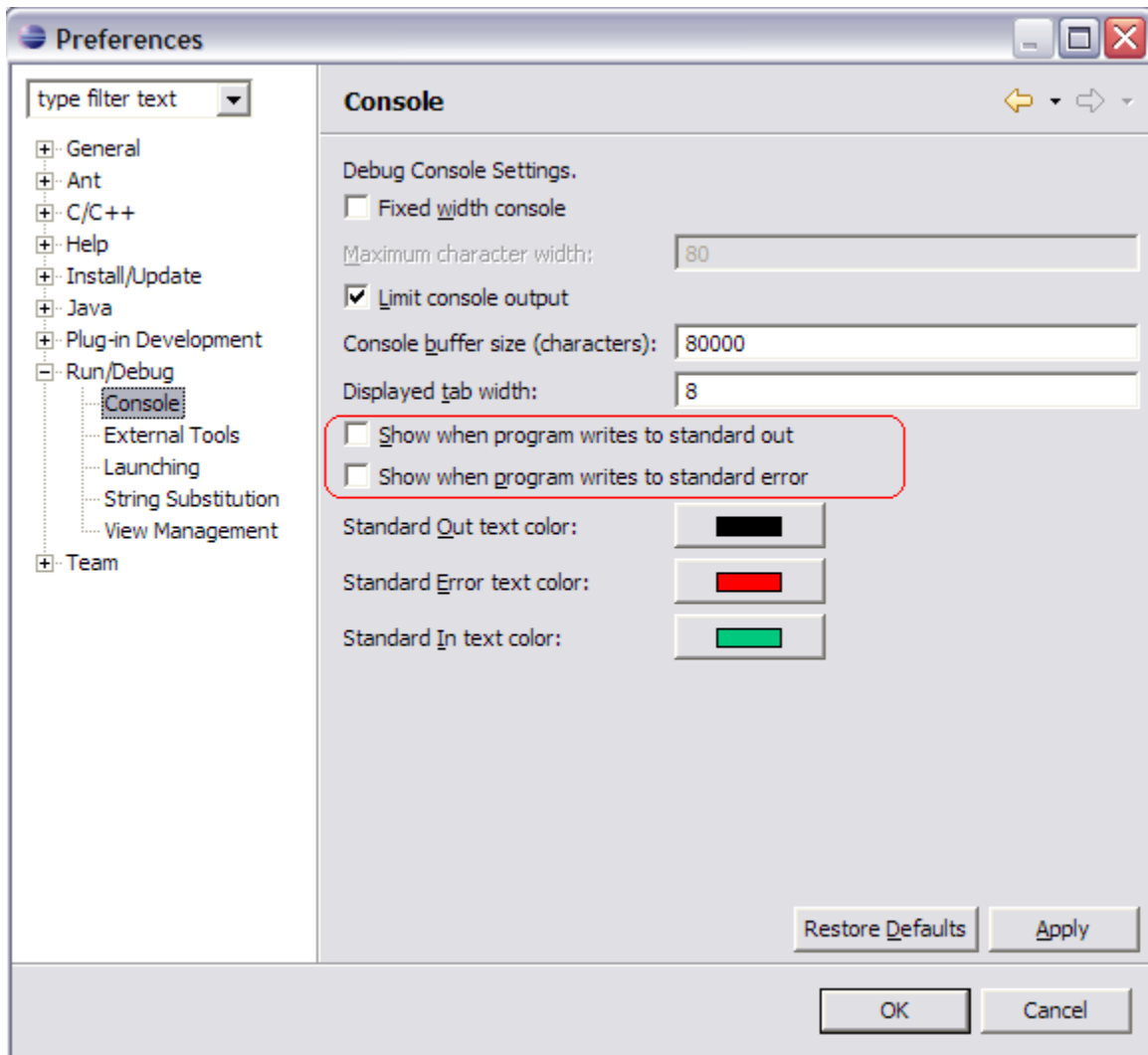
Default GDB Executable: This is the default GDB executable. The selected GDB should match the target CPU and should either come from the installation CD or be compiled from the sources provided.

2.2 Other Useful Preferences

2.2.1 Console

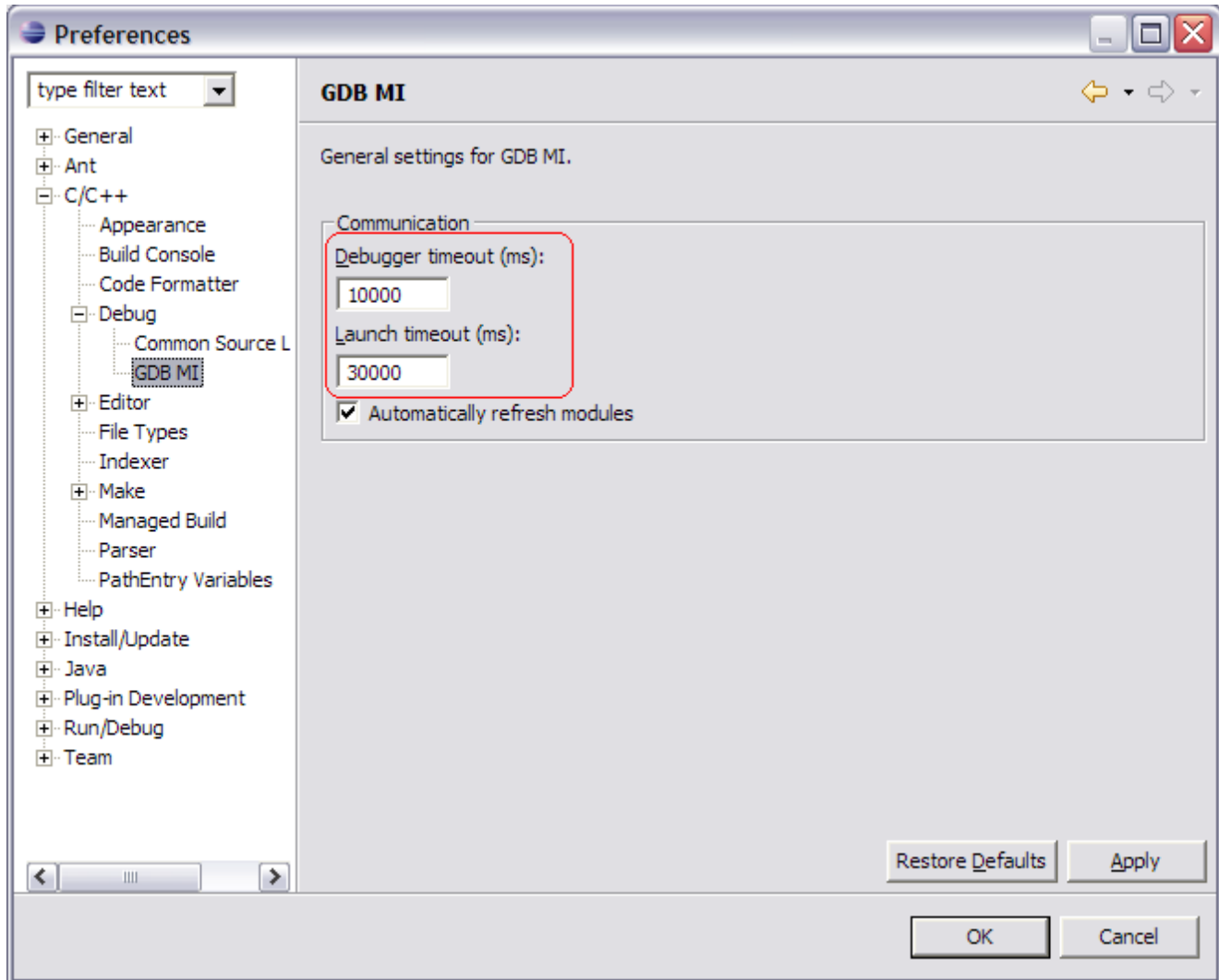
Note: These preferences are displayed only when the Preferences dialog is opened manually; the dialog opened by the cheat sheet does not contain these preference pages.

Eclipse console windows can be configured to pop up whenever there is new output displayed. This can become aggravating as GDB frequently produces output that is not interesting enough to warrant a distraction. This behavior can be disabled by unchecking the “Show when program writes to standard out/error” options on the Run/Debug->Console preferences page.



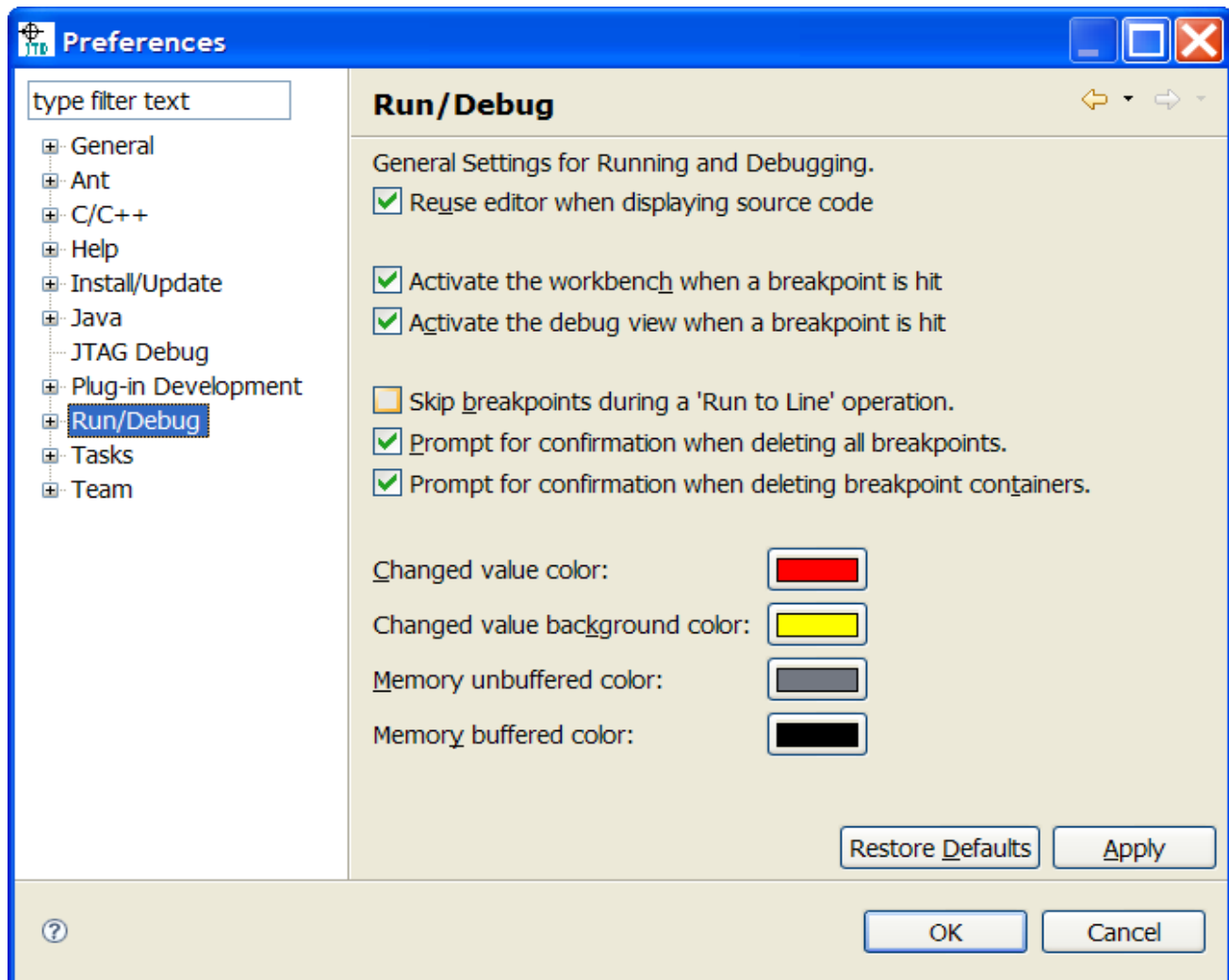
2.2.2 GDB Timeouts

When sending remote commands to the JTAG probe, GDB uses a timeout value specified in the C/C++->Debug->GDB MI (machine interface) preferences pane. This value can be adjusted if GDB receives spurious timeout errors.



2.2.3 Source Editor Preferences

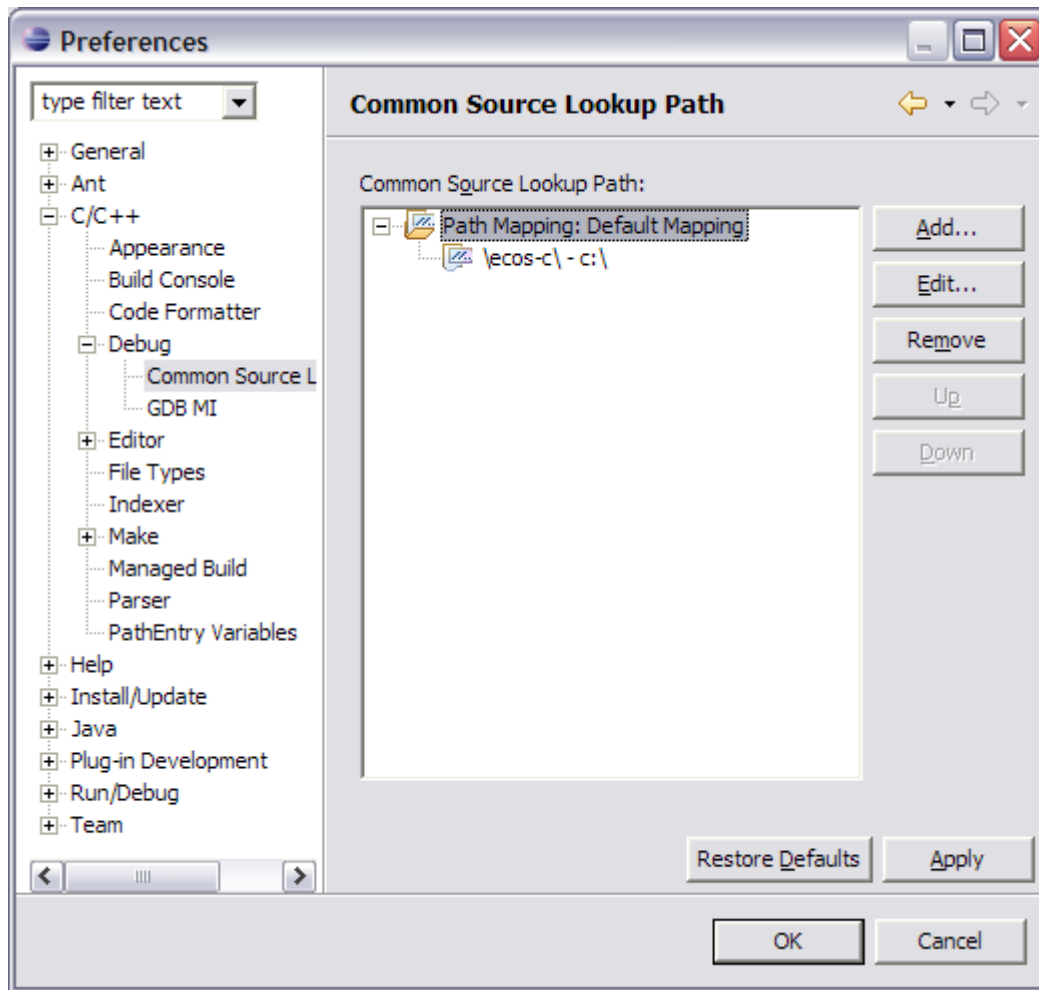
During debugging, Eclipse automatically reuses the same editor window when opening source files. This behavior can be changed on the Run/Debug preferences pane:



2.2.4 Path Mapping

In some cases, paths returned by GDB may not be understood by Eclipse because they do not resolve to files in the file system. For these cases, a path mapping can be created. The preferences window provides a way to specify a default set of path mappings under C/C++->Debug->Common Source Lookup Path.

Note: LinuxScope automatically takes care of path issues created by default Cygwin installations.



3 Working with LinuxScope-JTD

3.1 Launching Perspectives

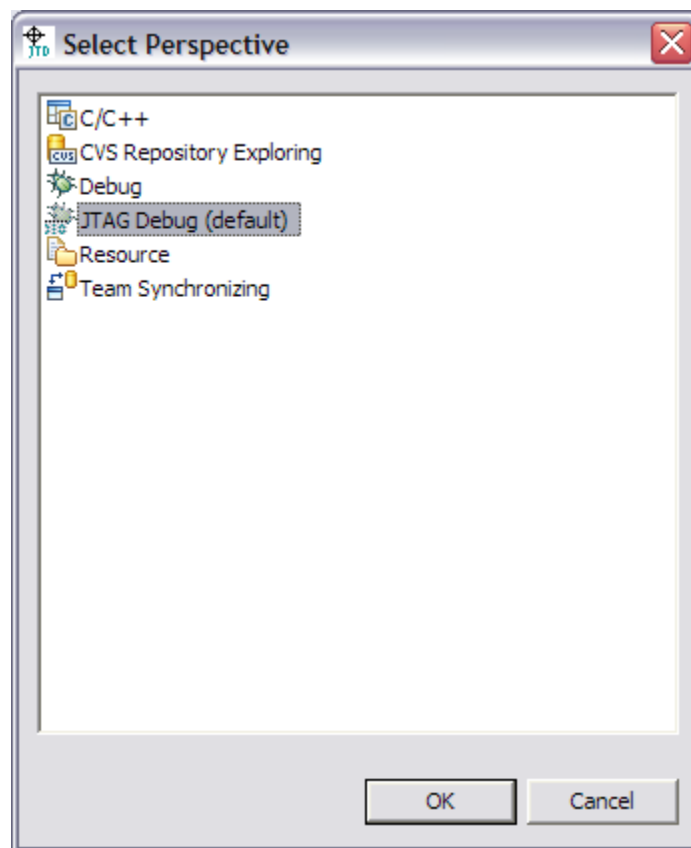
Perspectives in Eclipse are simply layouts of views and editors. While these can be customized, several have been provided for convenience. The “Open Perspective” button and menu item can be used to open perspectives.



The perspectives toolbar provides easy access to opening and switching perspectives:



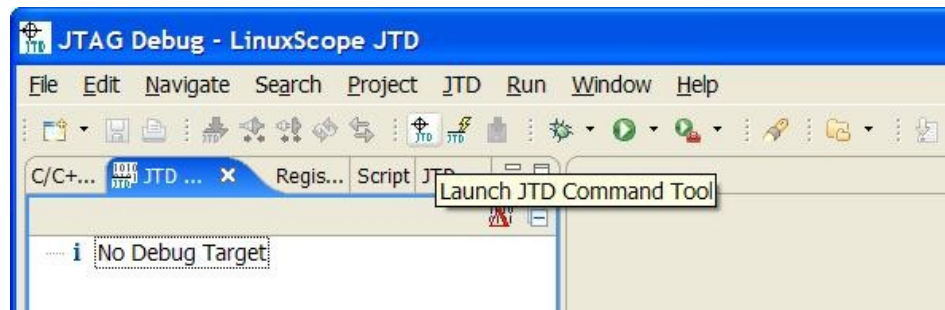
The “JTAG Debug” perspective provides a convenient layout of commonly-used debugging functions in addition to some JTAG-specific views.



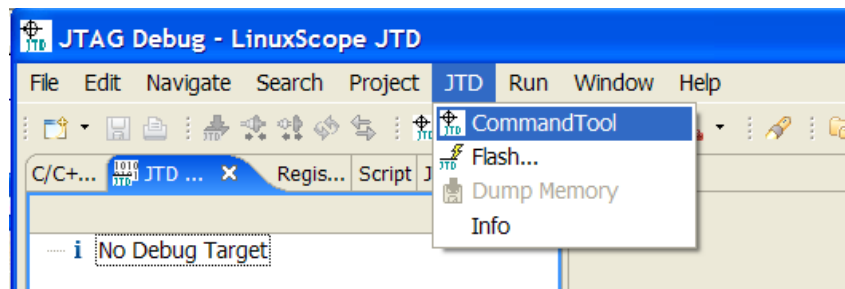
Perspectives can be customized by dragging views around and by using items on the Window menu. The default perspective configuration can be restored by selecting the Window->Reset Perspective option.

3.2 Command Tool

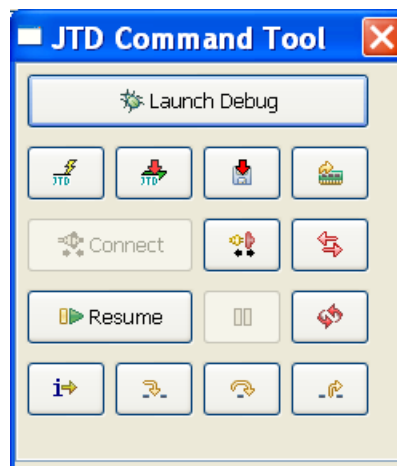
The JTD command tool offers an easy way to access key debugging functions from one toolbox. You can start it by clicking on the CommandTool button from the menu bar.



You can also start it from the JTD menu.



The toolbox will launch. If you are not connected to a target most of the buttons will be grayed out.



The Command Tool buttons include:



Launch Debug: Launch debug sessions



Memory Dump: Dumps the target memory to a file



Load Memory: Load a binary file to target memory.



Load: Loads the binary image into the target via the JTAG probe



Flash: Launch the Flashing Utility



Instruction Stepping Mode: Step instructions rather than source lines; this must be selected if there are no sources or if the stack is not valid.



Step Into: Step into function or branch instruction



Step Over: Step over statement or instruction



Step Out: Continue execution until function exits



Run/Resume: Resumes target execution



Suspend: Suspends target execution



Connect: Connect to remote target; this is used if the session was launched without the “Attach to JTAG target on startup” option



Disconnect: Disconnects from the JTAG probe but does not kill the debug session



Reset: Resets the target connected to the JTAG probe via the GDB “load” command



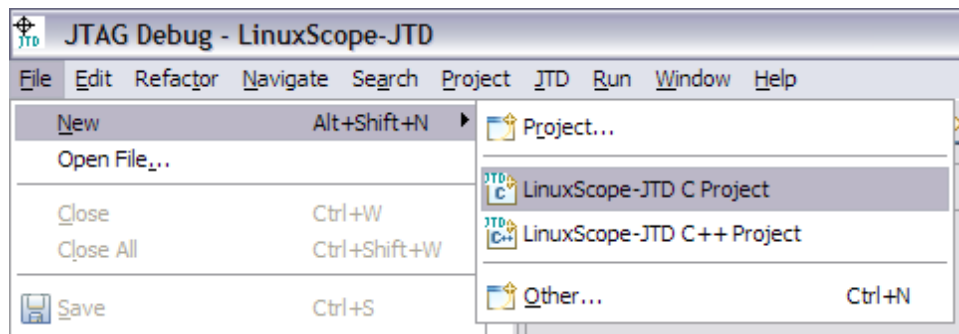
Sync: Synchronizes LinuxScope, GDB, and the JTAG probe; useful if the probe was accessed directly via Telnet

3.3 Creating a Project

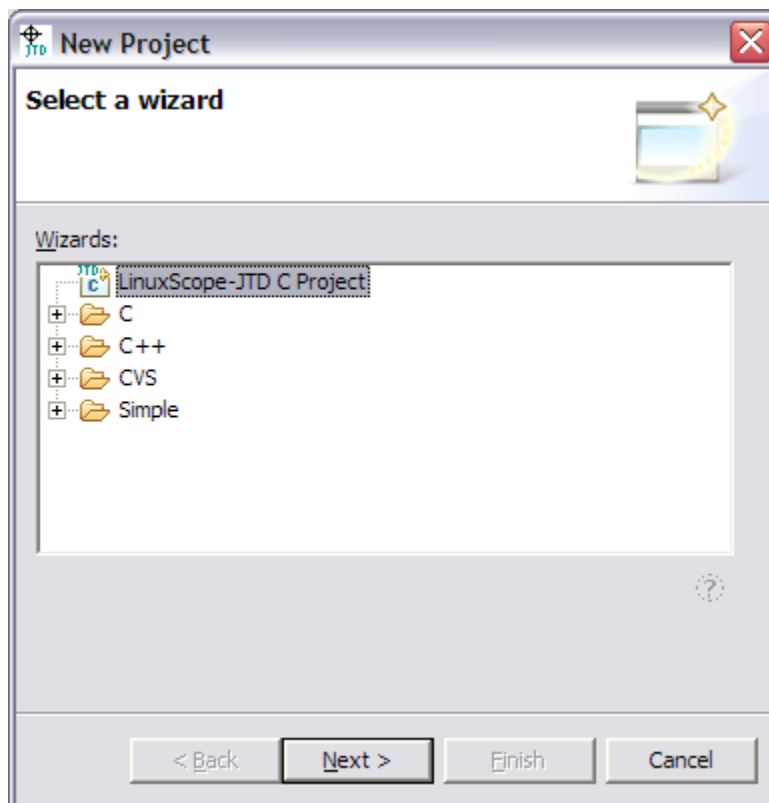
Before performing any debugging, a project is required. An Eclipse project is a file system directory/folder that contains files related to a coding/debugging “project”. Even if no sources and binaries are to be stored in Eclipse, a project is still required to save breakpoints, launch configurations, and so on.

CDT provides several project types. These project types help build code, by means of “standard” and “managed” makefiles. Such projects can be used in LinuxScope debugging sessions.

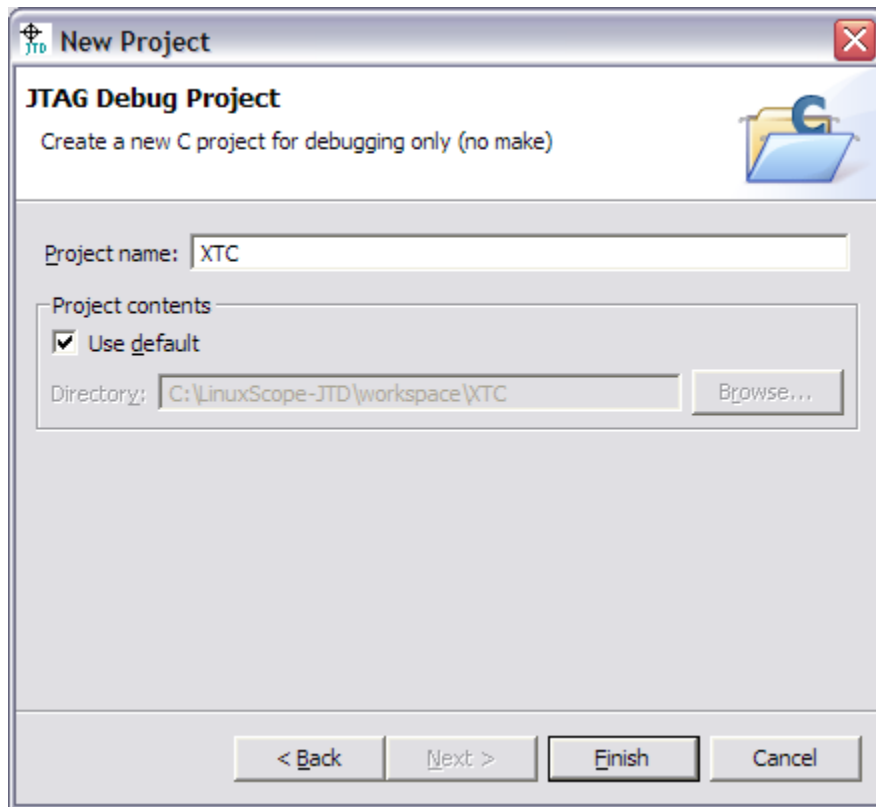
Setup of a complex CDT project can be bypassed if CDT is not used to build the binary images used for debugging. To create an empty project suitable for debugging, select “File->New->Project...” or “File->New->LinuxScope-JTD C Project...”.



Select a project type if asked and select “Next”:

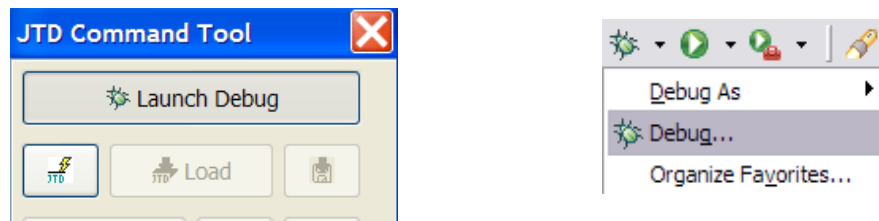


Type a project name and choose “Finish”:

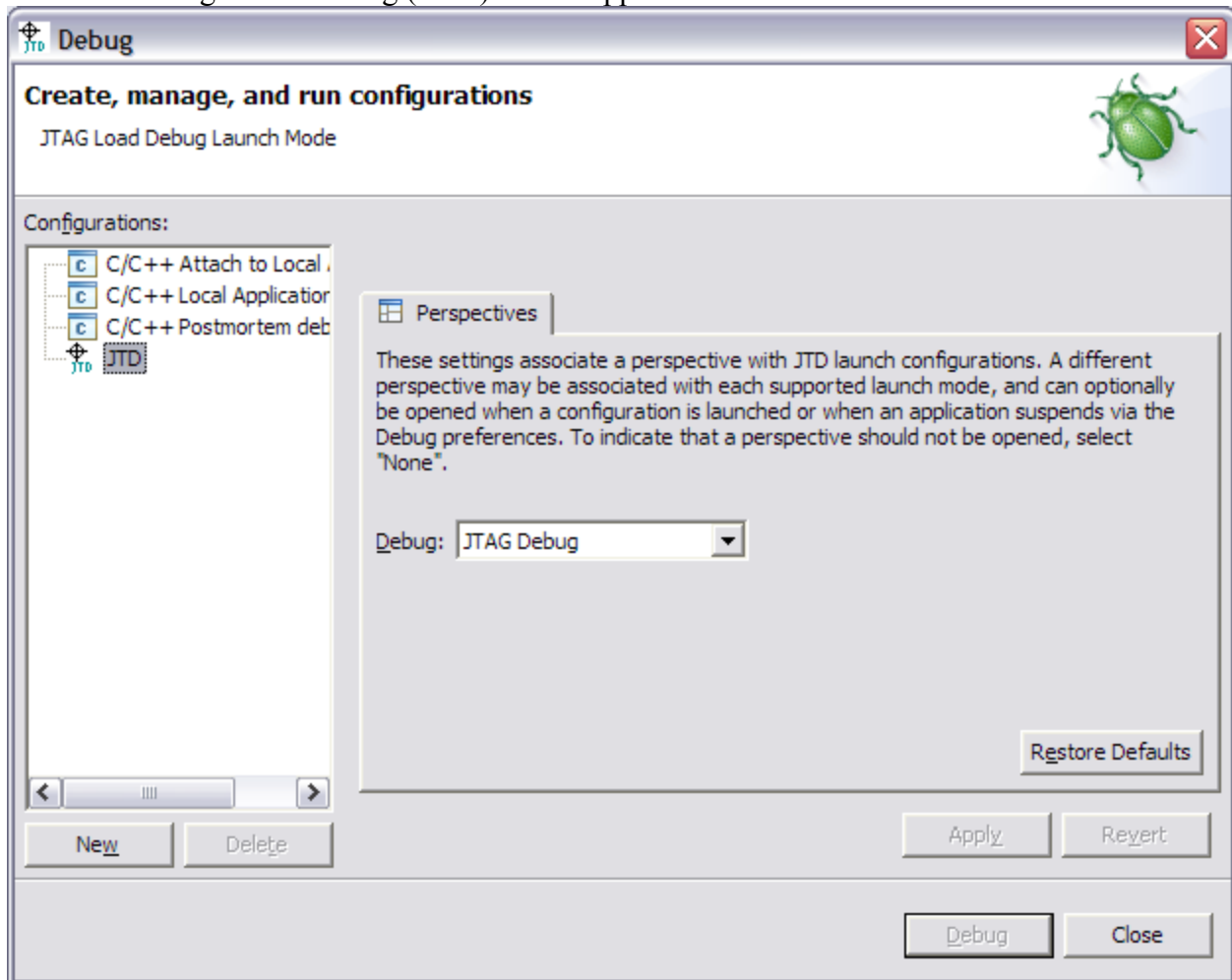


3.4 Launching Debug Sessions

To begin a debugging session, select “Launch Debug” from the CommandTool. You can also select “Debug...” from the debug dropdown, or “Run->Debug...” from the main menu:

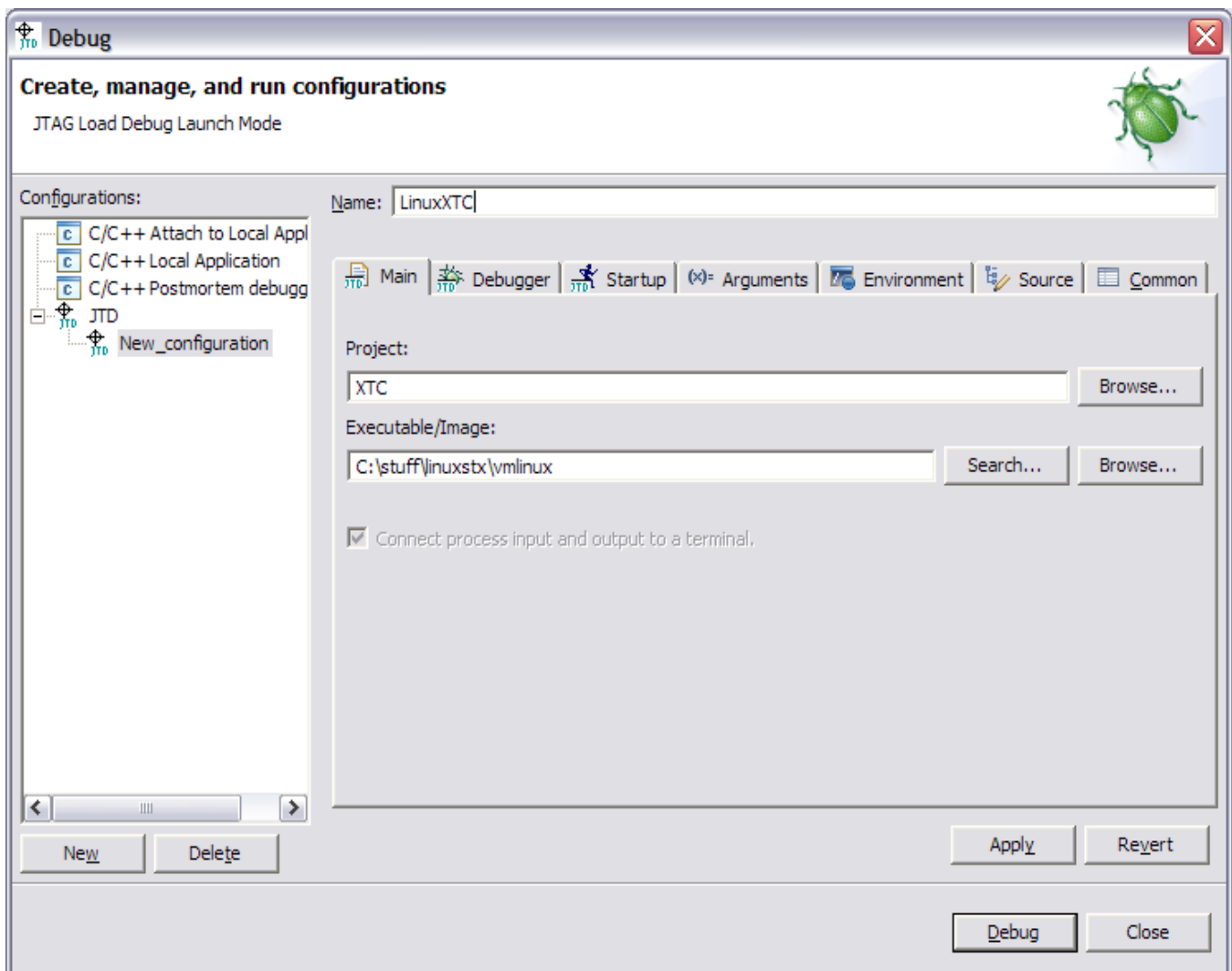


The Launch Configuration Dialog (LCD) should appear.

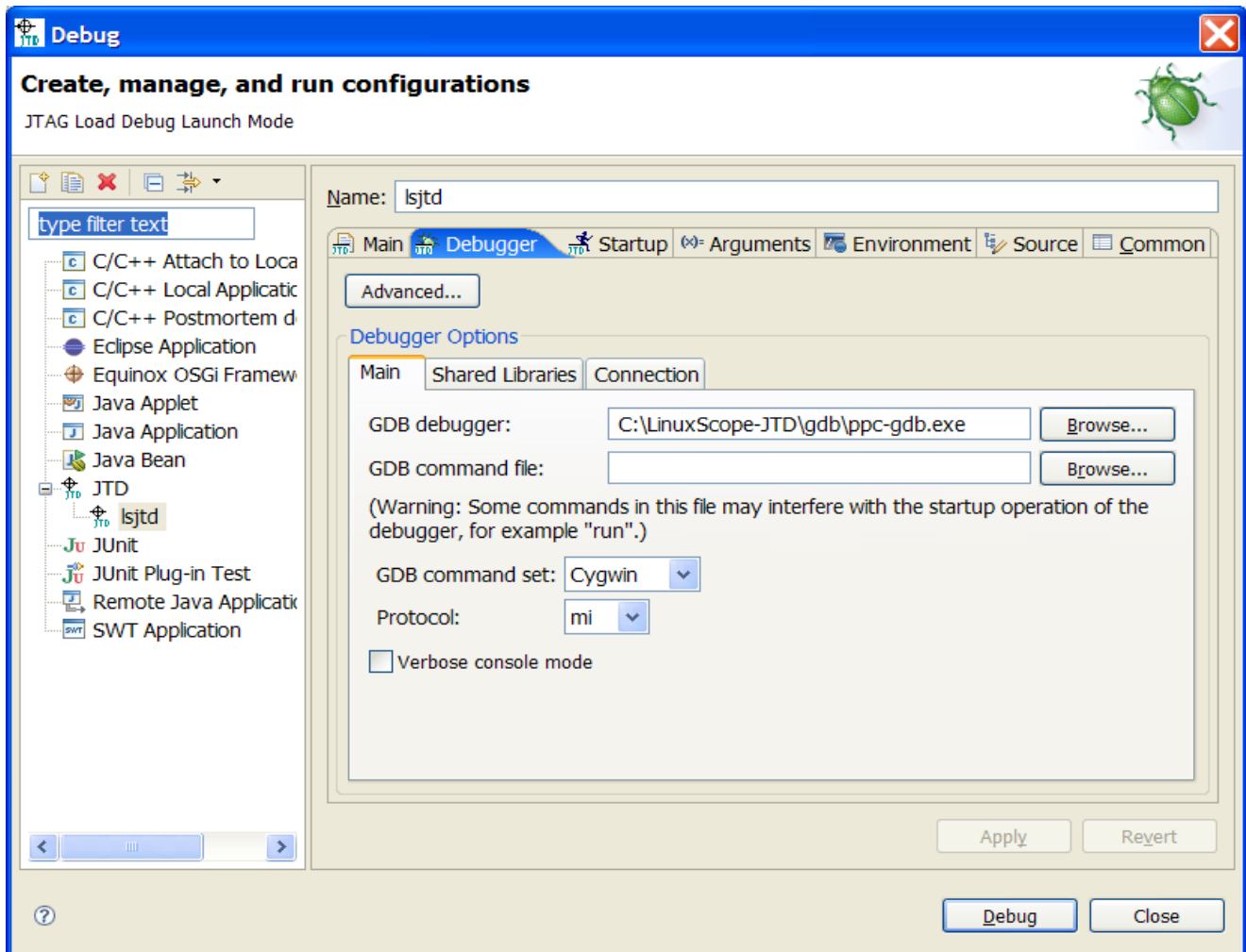


The JTD launch configuration type has several launch options that are specific to JTAG debugging. To create a new launch configuration, select the “JTD” type and then press the “New” button. This creates a new set of launch parameters, which is independent of other launches. Tabs allow the launch to be configured:

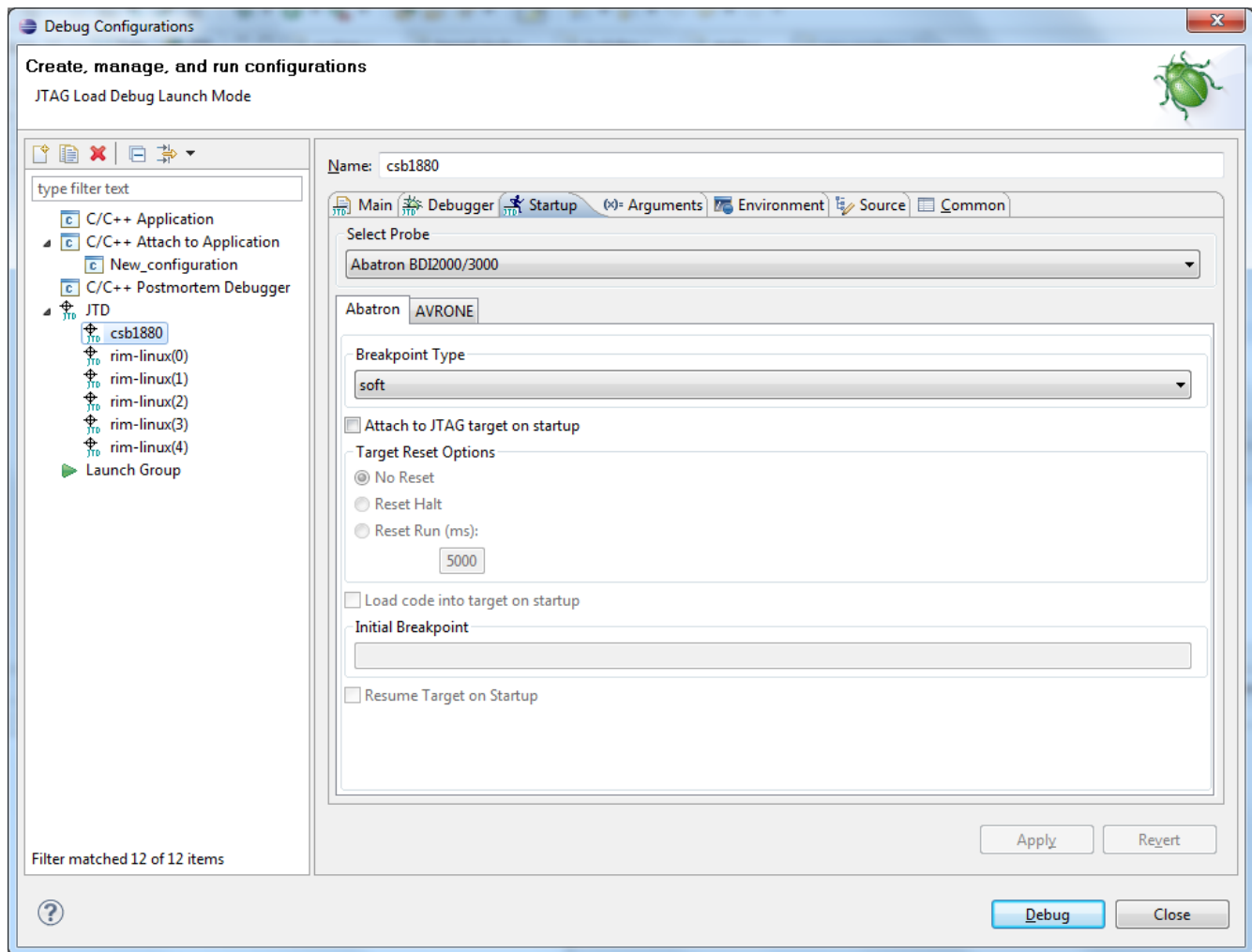
Main Tab: This allows the project and executable image to be specified. Configurations do not require executables, but use executable images to resolve symbols. If a project and executable were selected prior to opening the launch dialog, these should automatically be populated in the fields. Executables can be located anywhere in the file system, not just inside the project.



Debugger Tab: This tab allows the GDB executable and remote settings to be overridden. This is initially populated from the “Preferences” page. The “Shared Libraries” tab allows shared library handling to be adjusted. Click on the “connection” sub-tab to see the TCP/IP settings.



Startup Tab: This tab contains several options for controlling startup behavior when the debug session is launched.



LinuxScope can also connect to the AVRONE debugger and you can use the “Select Probe” drop down box to select which probe you want to use.

“Breakpoint Type” selects a breakpoint setting for the probe. Hard breakpoints make use of hardware registers and the number of simultaneous breakpoints is usually quite limited. Soft breakpoints are more flexible, but rely on trap instructions and therefore cannot be used when debugging flash images or when the IVT is not set up. While using the soft breakpoint mode, it is still possible to set hard breakpoints explicitly.

If “Attach to JTAG target on startup” is checked, GDB connects to the probe during the launch. If it is not checked, the user can connect to the debugger at a later time.

The “Target Reset Options” group controls whether the target is reset during the launch sequence. If so, the target can be run for a specified amount of time to allow the target to initialize.

If “Load code into target on startup” is checked, GDB will be used to load the program into the target during the launch process, after reset (if any). This can also be done manually.

If specified, the “Initial Breakpoint” is used to stop the target when execution reaches a symbol or address (prefixed with '*'). Typically, 'main' is used in C applications, however sometimes more flexibility is warranted when debugging embedded code.

If “Resume Target on Startup” is selected, target execution will resume once the launch is configured. Otherwise, the target will remain suspended after the launch.

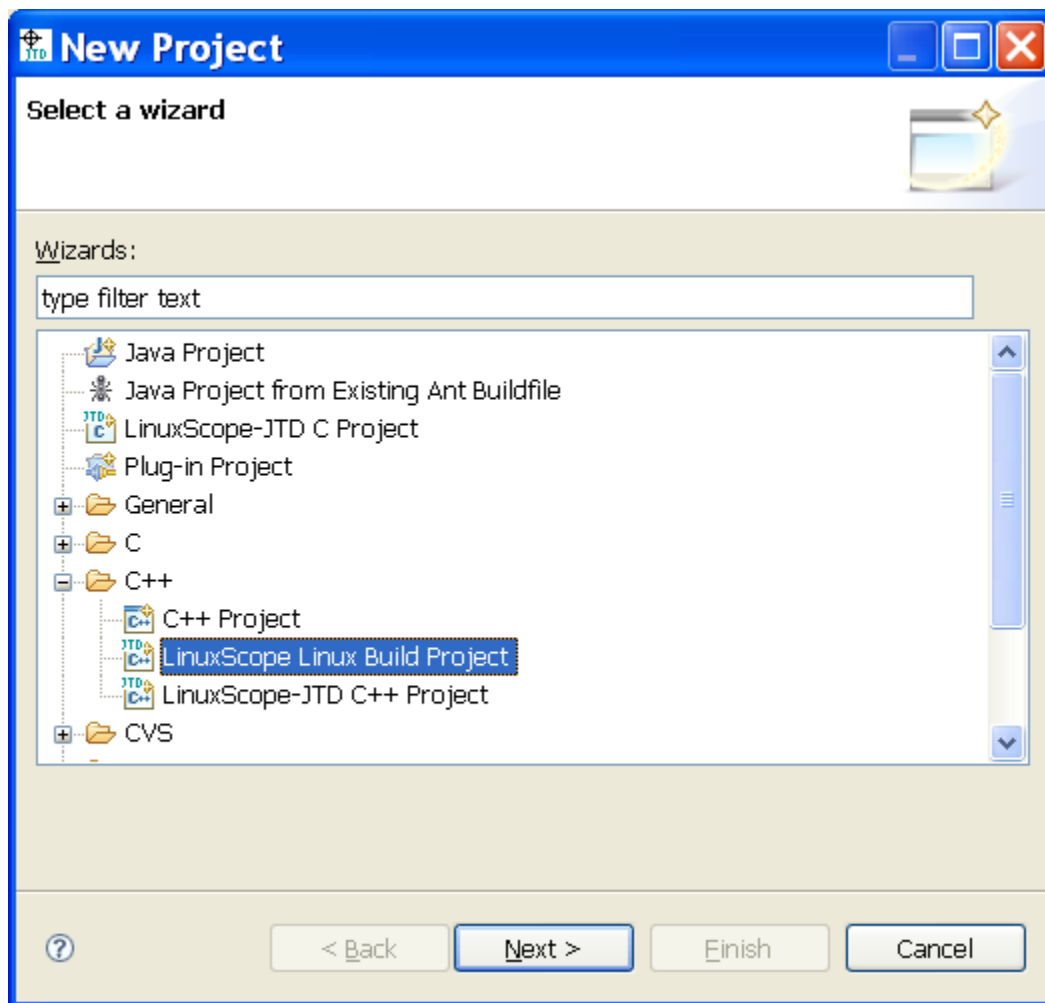
Arguments/Environment Tabs: Used to pass arguments and environment variables to the target program. These are not typically used with remote targets.

Source Tab: Specifies path entries and substitutions for source lookup. This is very useful for sources that exist outside the project.

3.5 Creating A Linux Build Project

LinuxScope can be used to build a Linux kernel or any GNU Make project. The steps to creating a project for building sources is similar to that of creating a project. But some additional parameters will need to be specified so that the LinuxScope environment can recognize and use the tool chain.

Start by first making a working copy of your Linux Kernel or GNU Make sources. Then from LinuxScope select “file->new->project” expand C++ and select “LinuxScope Linux Build Project”



Click “Next”. In the next window you will need to fill out several settings.

New Project

Linux Build Project
Create a new C project for a Linux Kernel

Name of this Linux Kernel Project

☐ Create an empty Project or uncheck and browse to existing Kernel directory

Location of Linux Kernel

Select the Tool Chain Directories

C:\cygwin\opt\yeldk\bin	
C:\cygwin\opt\yeldk\usr\bin	
C:\cygwin\bin	
C:\cygwin\usr\sbin	

☒ Add Variables for Linux Build

CROSS_COMPILE=

ARCH=

☒ Add Make Targets

Make Target

all	
clean	
distclean	

Start by specifying a project name for this Linux Build Project.

Uncheck the checkbox “create an empty Project ...” and click “Browse” to the top level source directory of your Linux Kernel.

Next click “browse/add” and select your tool chain directory, this is the directory(s) where your GCC and other tools required to build the kernel reside. When you install your tool chain It should give you details on which directories to include. If you are using cygwin for your build environment you will also need to include the /bin/ and /usr/bin/ directories under your cygwin root directory so that LinuxScope can find the build environment.

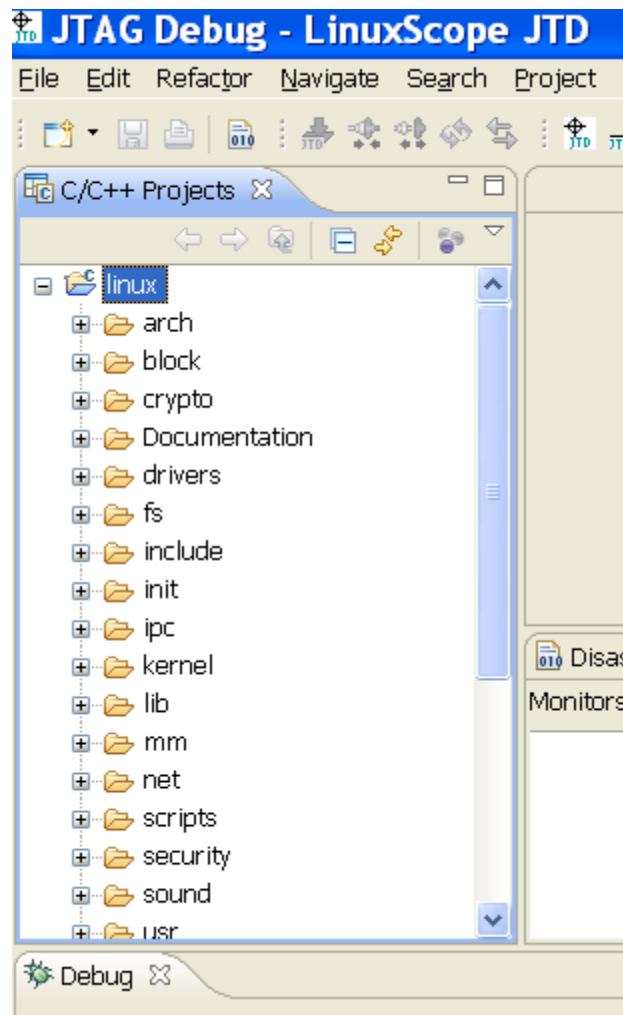
Select the “Add Variables for Linux Build” and specify the CROSS_COMPILE and ARCH system variables, your tool chain documentation should tell you what these values need to be. For other GNU make projects these variables may not be needed.

If you want to add make targets to build then check the “Add Make Targets” and add/remove the make

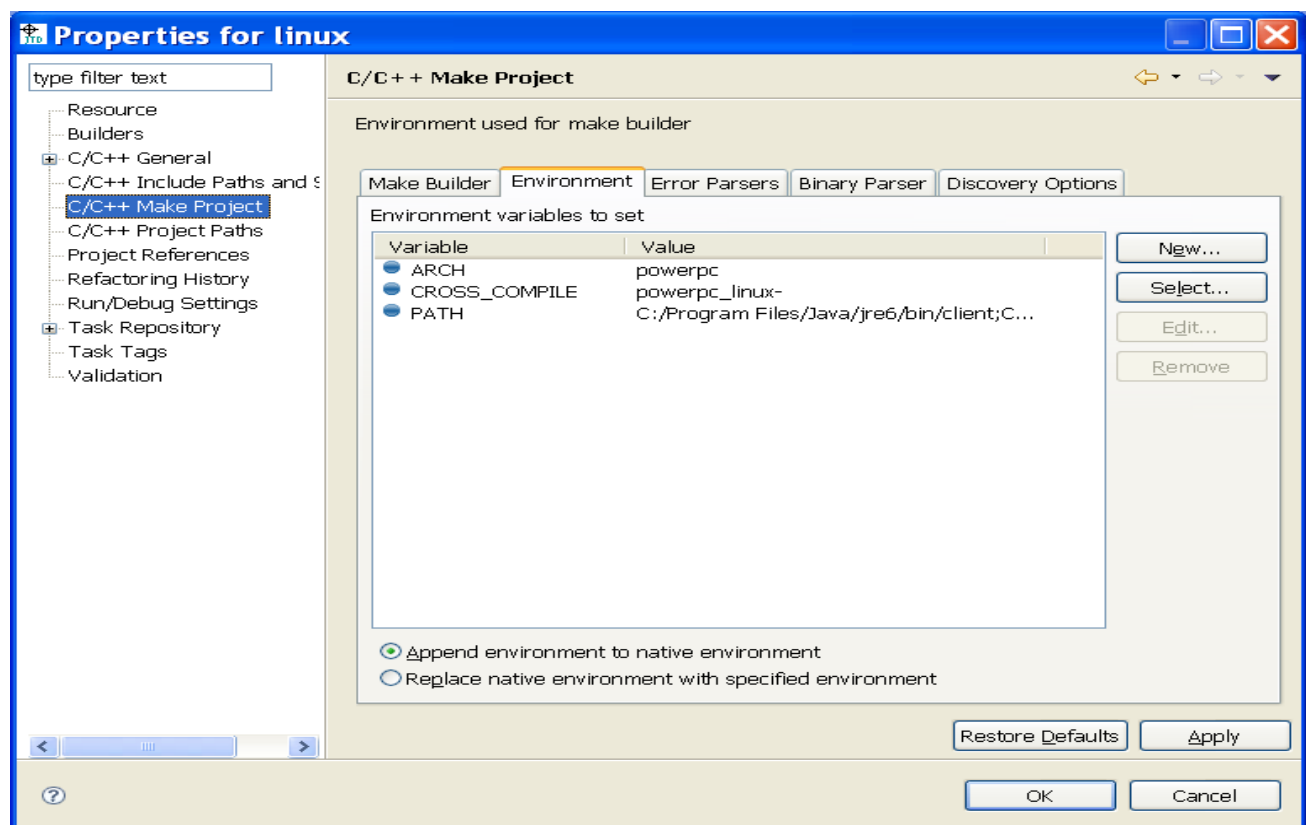
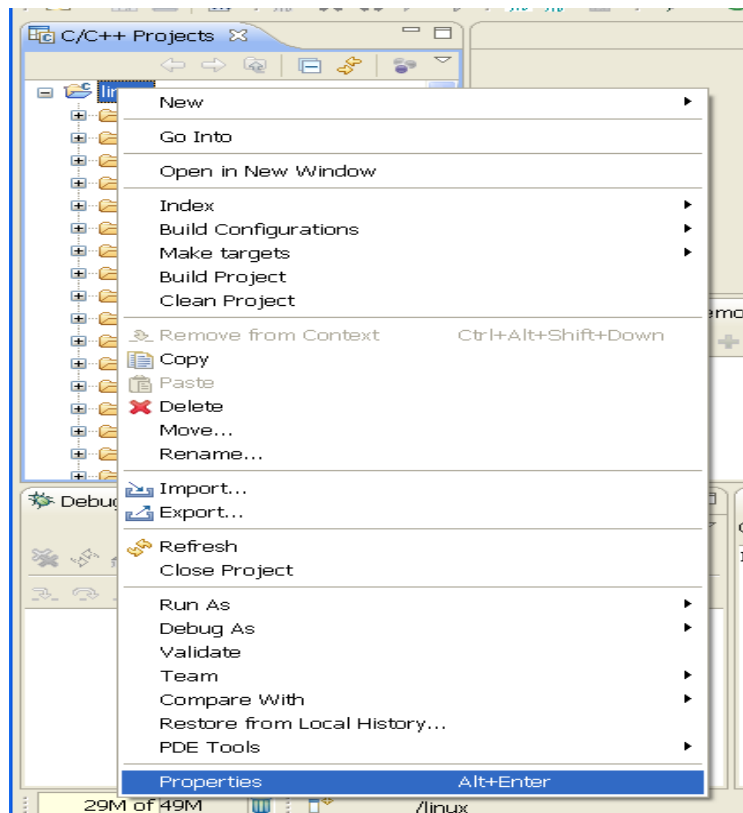
targets you wish to include.

Once you have setup this page click “Finish”.

A new project with your chosen name is created and displayed in the “C++ Projects” view.

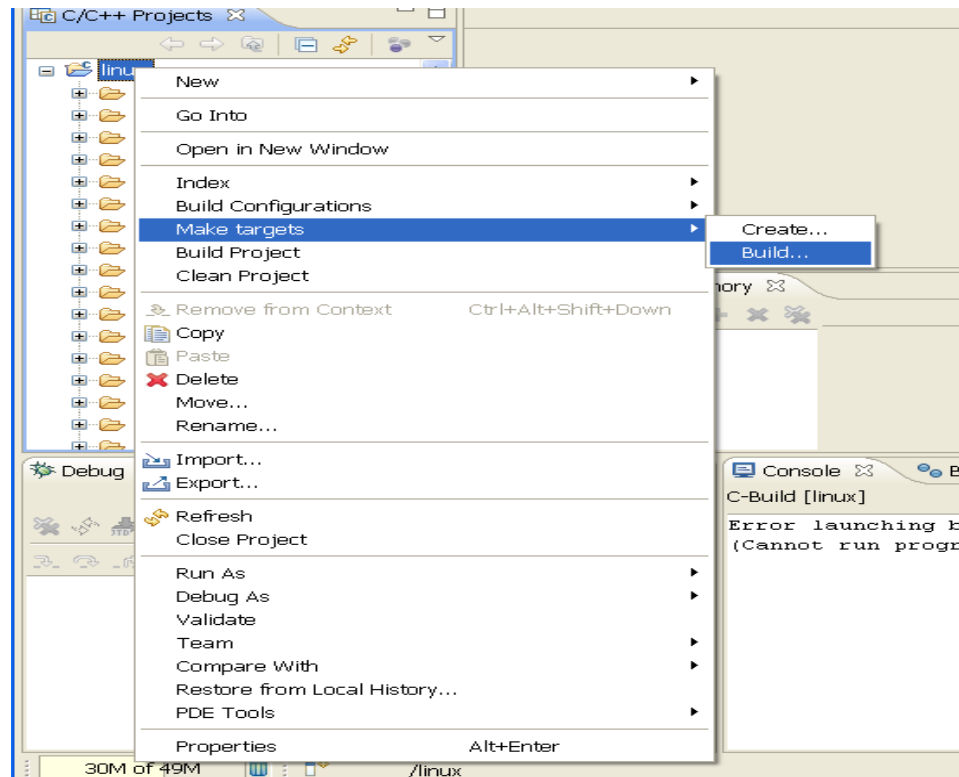


To edit your environment settings right click on the project and select “properties”. From the pop up select the “C/C++ Make Project” and then select the “environment” Tab.

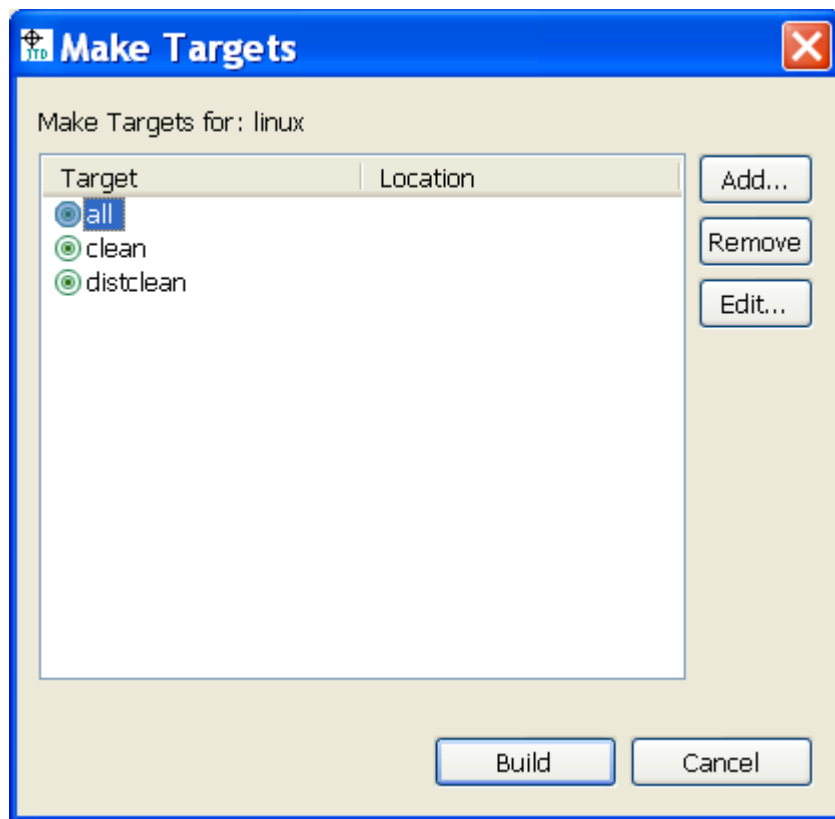


Add remove any environment variable that needs to be set to make your project build.

To build one of the make targets you have included right click on the project and select “make targets->Build”



Select the target you want to build or add a new target then select the “Build” button.



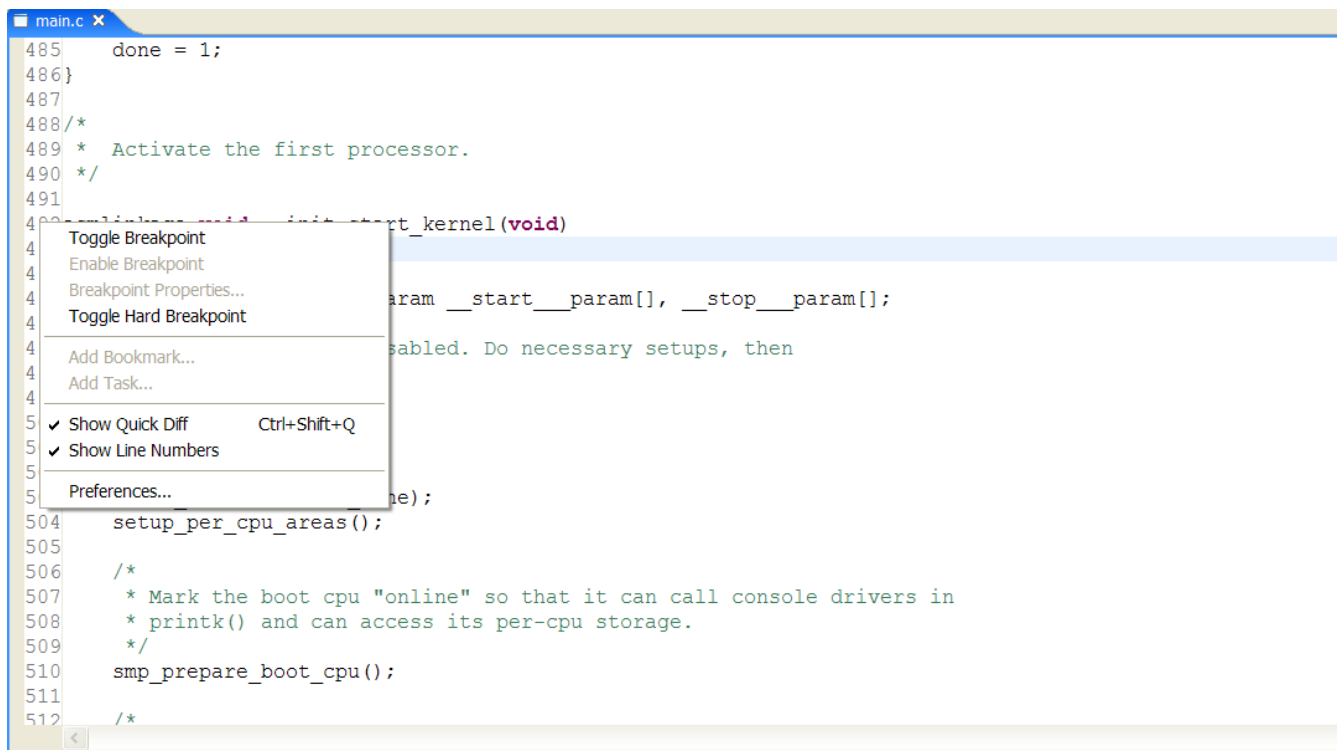
4 Source Debugging

4.1 Connecting To Target

If you have not chosen to automatically connect to the target in your launch configuration you will need to connect before you can debug. Pressing the “connect to target” button in the Command Tool will establish a connection to your JTAG Probe (Refer to Section 3.2 for Command Tool).

4.2 Setting Breakpoints

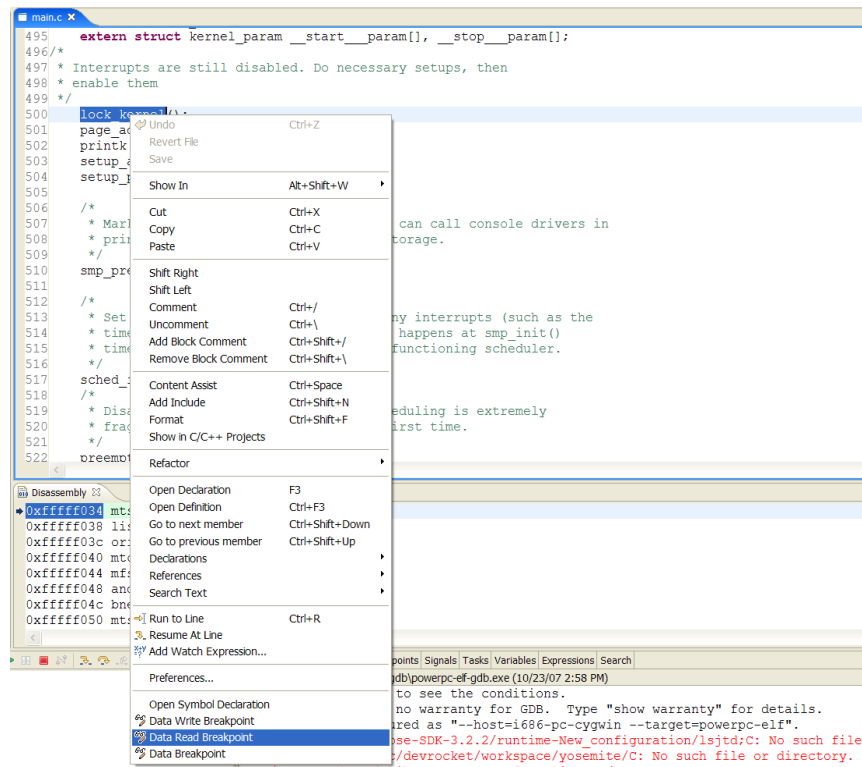
Breakpoints can be set from the source window by right clicking on the white space next to the source line or on the line number of the source line.



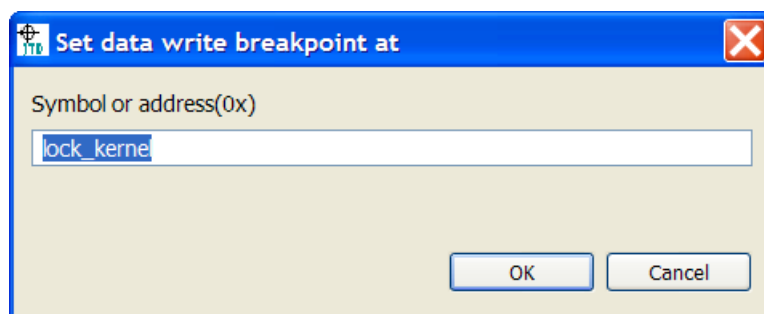
NOTE: The option to set a hardware breakpoint is only enabled if you are connected to the target.

4.3 Setting Data Breakpoints

Breakpoints can be set from the source window by right clicking on the white space next to the source line or on the line number of the source line.

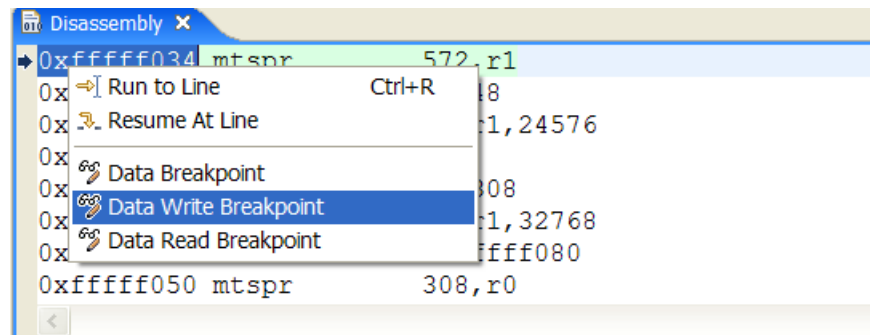


There are 3 types of data breakpoints that can be set. Data Write Breakpoint, Data Read Breakpoint and a Data Access Breakpoint. You can choose to set any type of Data Breakpoint on a symbol or global variable. Select the function or variable name and select the option to set the breakpoint type you would like. A dialog box will appear asking you to confirm your selection.



At this point if you want you can change the symbol name to set the breakpoint on. You may also choose to select an address location. If you choose an address location you must enter the full string as "0xffffffff", a total of 10 characters including the "0x". Once you have made any changes click on "OK" to set the breakpoint. Provided the symbol is found, the breakpoint will be set and you can view it from the breakpoints view. To remove the breakpoint you should use the breakpoints view (Section 5.8).

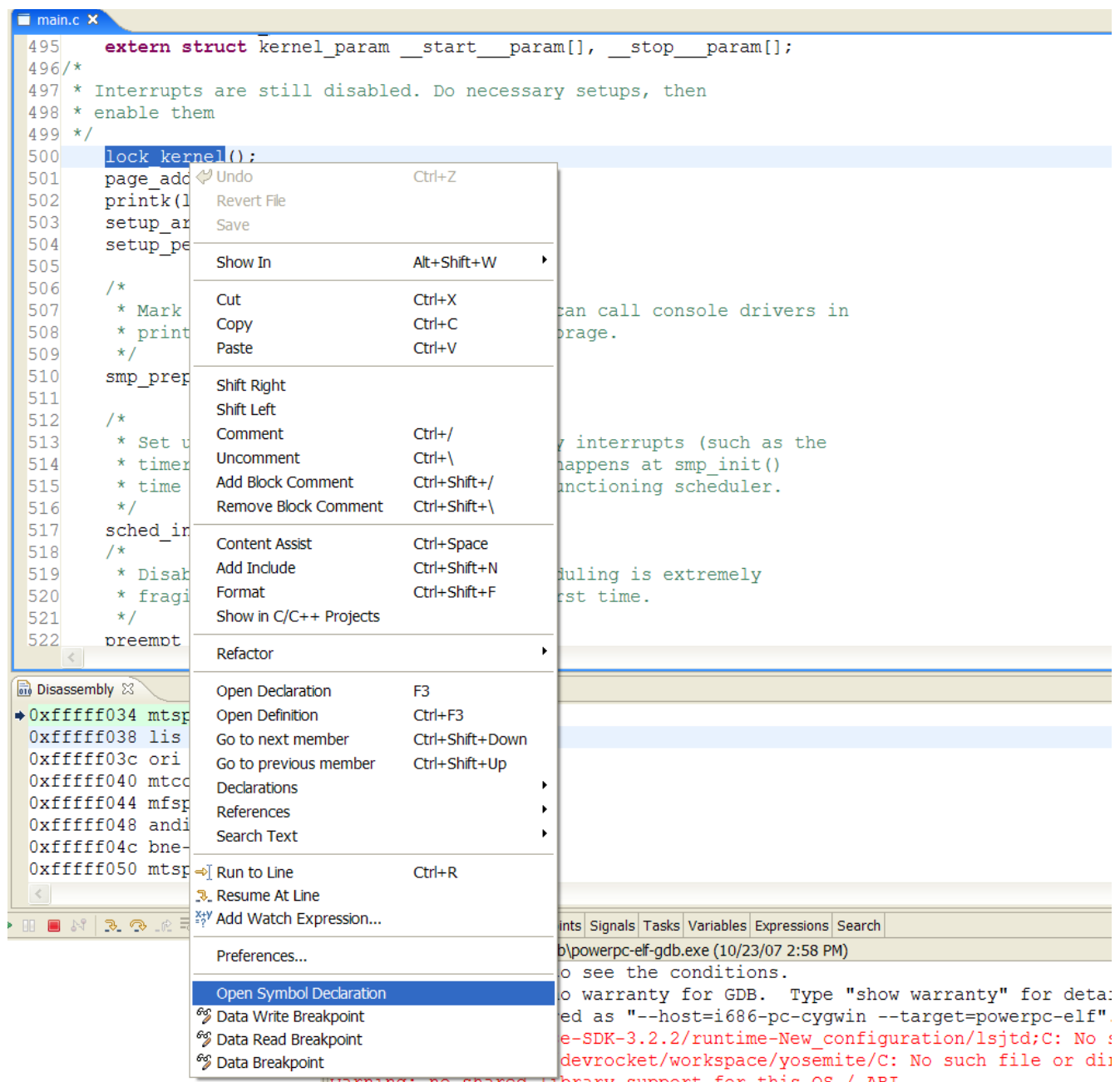
You may also choose to set data breakpoints from the disassembly window. Select the address where you want to set a data breakpoint and right click to open the selection menu.



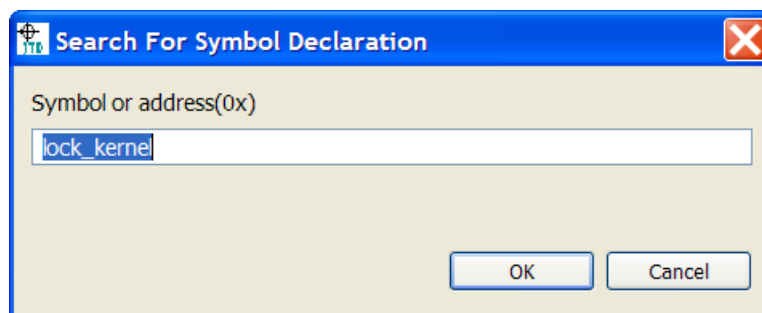
NOTE: The options to set data breakpoints are only available when you have launched a valid debug session and are connected to the target.

4.4 Finding Function and Variable Declarations

You can open function and variable declarations directly from the source window by selecting the function or variable in question and selecting the “Open Symbol Declaration” option.



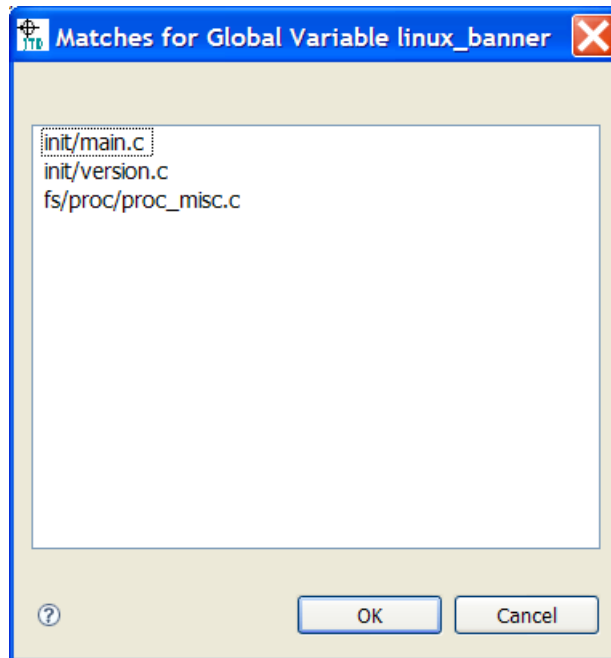
A dialog box will appear asking you to finalize your selection.



At this point you can change the symbol name to set the breakpoint on if desired. You may also choose

to select an address location. If you choose an address location you must enter the full string as “0xffffffff”, a total of 10 characters including the “0x”. Once you have made any changes click on “OK”. Provided the symbol or address is found, the source containing the function declaration or address location will open and the line number will be highlighted.

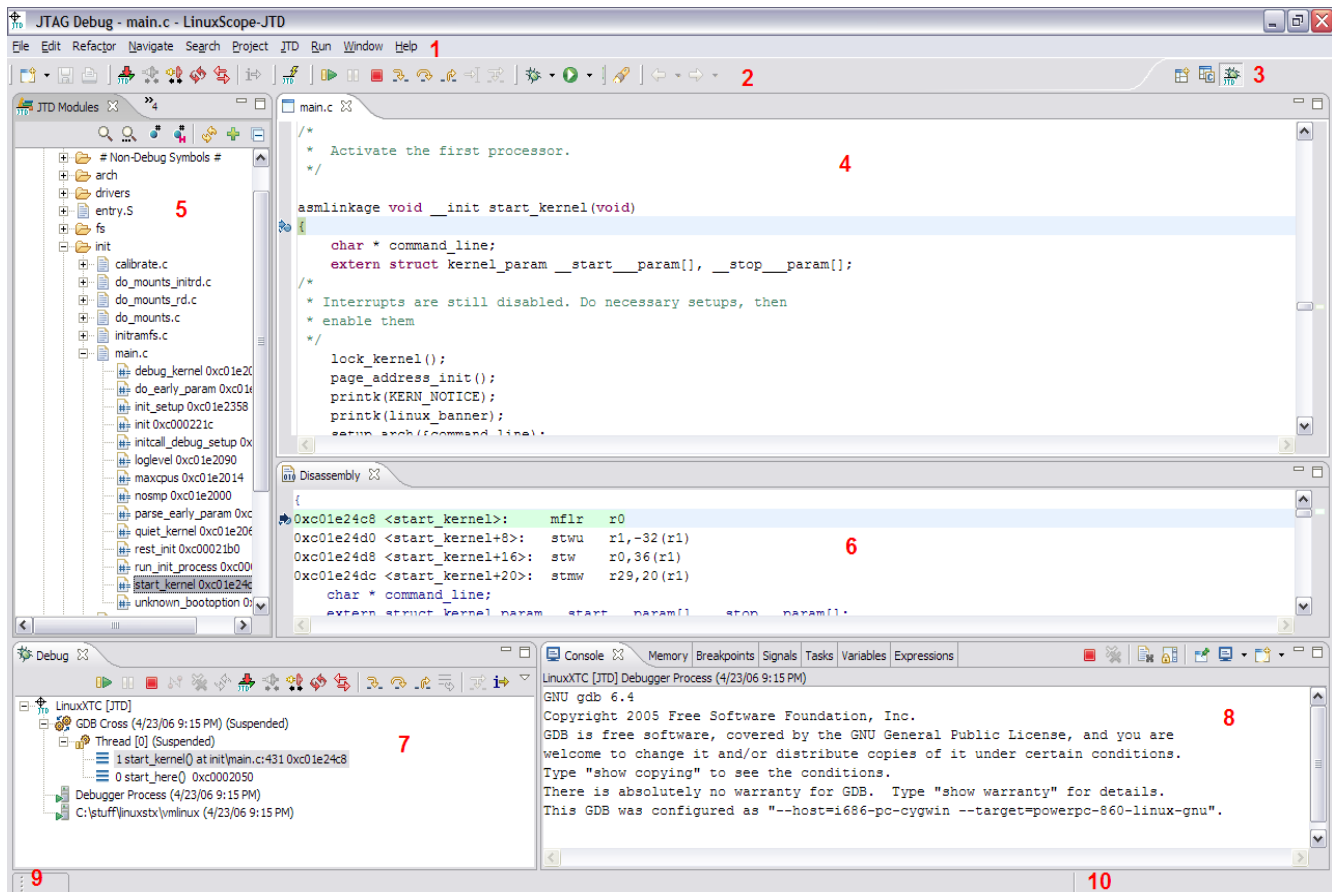
If the symbol selected is a variable then another selection box will open asking you to select a source file match for the given variable.



Make a selection and click “OK”. You will be taken to the selected source file and line number where the variable is declared.

NOTE: The option to open symbol declarations is only available if a valid gdb session has been launched.

5 JTAG Debug Perspective Tour

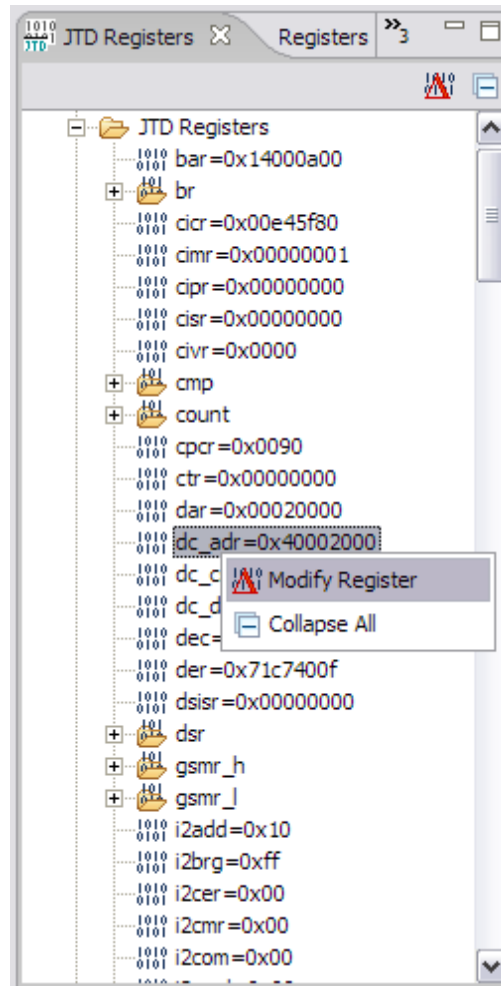


The JTAG Debug perspective contains several features:

1. Menu Bar: pull-down menus
2. Toolbar: Quick access to common operations
3. Perspective Chooser: Open and switch perspectives
4. Editor Area: Used for source code displays
5. Left View Area: Contains registers, modules, script, and project views.
6. Disassembly View: Shows current assembly instructions
7. Debug View: Shows current target processes and stack frames
8. Debug Views: Contains several debug views, including console, memory, and variables
9. Quick-View Panel: Drag views here for quick access
10. Progress Monitor: Displays status and progress information.

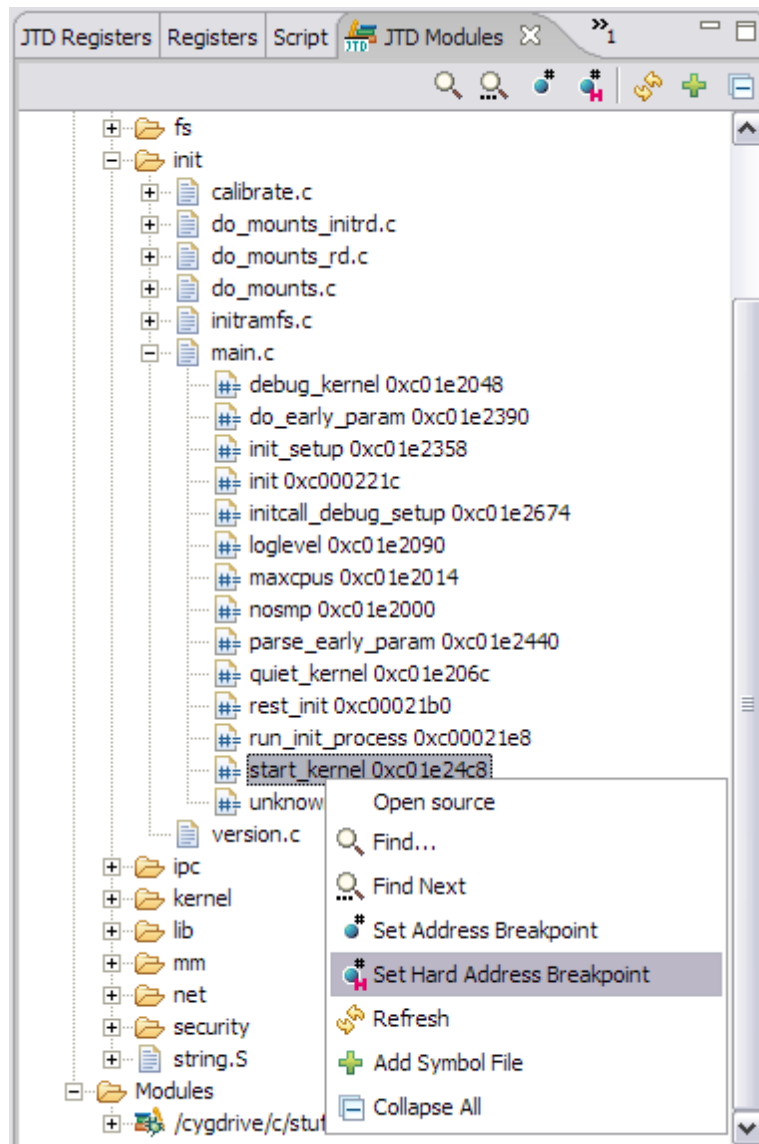
5.1 JTD Registers View

Within the JTAG Debug perspective, the JTD Registers view displays registers that GDB does not know about, but that are listed in the probe's register definition file. To modify a register, right click:



5.2 JTD Modules View

The “JTD Modules” view provides a tree of all modules and symbols loaded into GDB. This allows for symbol table searches, source location, and breakpoint identification.



The “JTD Modules” view provides the following actions:

Open Source: Opens the selected source file in an editor window, or opens the source file containing the selected symbol.

Find...: Searches for a text string in the sources and symbols trees, starting from the root.

Find Again: Repeats a search started with “Find...”, starting from the current selection.

Set Address Breakpoint: Creates a breakpoint on the selected symbol, using the probe breakpoint mode.

Set Hard Address Breakpoint: Creates a hard breakpoint on the selected symbol, regardless of the probe breakpoint mode.

Refresh: Updates the tree with regard to symbols loaded into GDB. This is useful if the GDB console has been used to modify the symbol table.

Add Symbol File: Allows an additional symbol file to be added to the address space. This is useful for adding Linux kernel module information.

Collapse All: Collapses the entire tree.

5.2.1 Adding Symbol Files

The “Add Symbol File” action on the JTD Modules View provides a way to add symbols, such as those associated with Linux kernel modules.

The screenshot shows the "Add Symbol File" dialog box. It has a title bar with a heart icon and a close button. The main content area is divided into several sections. The first section is "Module_1:" with a text input field and a "Browse..." button. Below this is the "Segments:" section, which has a radio button selected and four text input fields labeled ".text", ".bss", ".data", and ".rodata". The "Map:" section has a radio button and a large text area. The "Arguments:" section has a radio button and a text input field. At the bottom left, there is a "ClearSymbols" checkbox and two buttons labeled "Prev Module" and "Next Module". At the bottom right, there are "OK" and "Cancel" buttons.

To add a module, the name of the module and the segment addresses are required. There are three ways to add the segment addresses:

1. Provide the addresses for the segments listed under the “Segments” radio button. The '.text' segment is required, the others are optional.
2. Paste in the map. This is in the format output by “insmod -m” (Linux 2.4 kernels only). Each line should have the form <segment> <size> <address> <alignment>.
3. Provide the arguments to the GDB “add-symbol-file” command directly. The format is: <.text address> [-s <segment> <address>] This can be automatically generated by Linux 2.6.8 and later kernels using a script like the following:

To clear the current symbol table before loading the current modules select the 'ClearSymbols' checkbox.

Up to 50 modules may be added. Module information will be stored in the project's configuration settings for future sessions. Prev Module and Next Module provide a way to iterate through the list of modules. Clicking OK will load the module's symbols and present a new Add Symbol File screen.

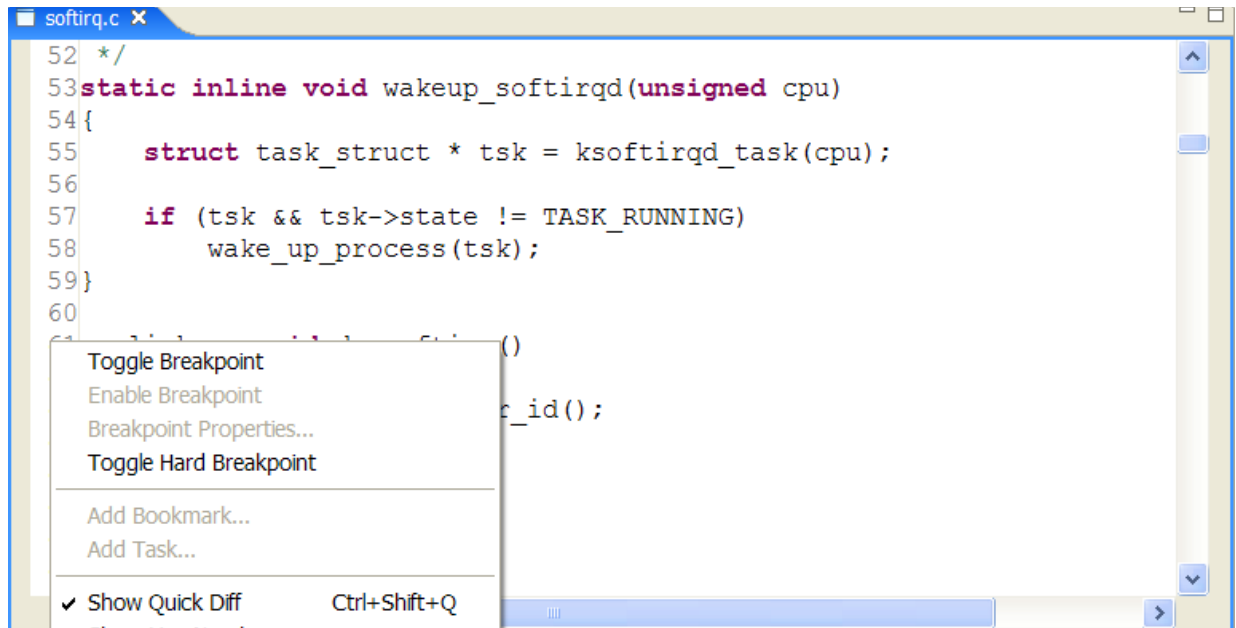
```
#!/bin/bash
#
# gdbline module image
#
# Outputs an add-symbol-file line suitable for pasting into gdb to examine
# a loaded module.
#
cd /sys/module/$1/sections
echo -n ` /bin/cat .text`

for section in `ls`; do
    if [ $section != ".text" ]; then
        echo -n " -s" $section ` /bin/cat $section`
    fi
done
echo
```

5.3 Finding Breakpoint Symbols

If sources are available to Eclipse, the GUI can be used to locate and set breakpoints.

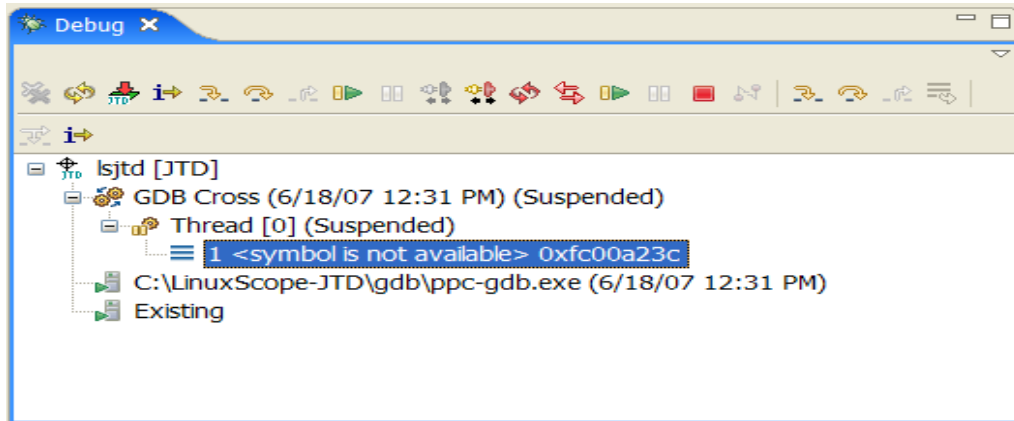
Breakpoints can be set directly on source lines. Breakpoints can be toggled by right-clicking to the left of source lines in the source editor:



The “JTD Modules View” can be used to set breakpoints. This uses GDB to identify sources and symbols.

5.4 Controlling Debugging with the Debug View

The debug view displays a hierarchy of selectable debug objects, and also provides a set of actions for controlling target execution.



Starting from the root of the Debug View tree, The hierarchy contains:

Launch Sessions: The debug session name

Processes: There is a process entry for the target, as well as helper processes.

Threads: Only one thread is supported by the hardware probe

Stack Frames: The symbol on the top of the stack appears at the top inside the window.

Many of the views and actions in Eclipse depend on the selection made in the debug view. For example:

The variables view displays variables for the current stack frame

The state of the debug actions depends on the selected target thread

The console view shows the input and output of the selected process

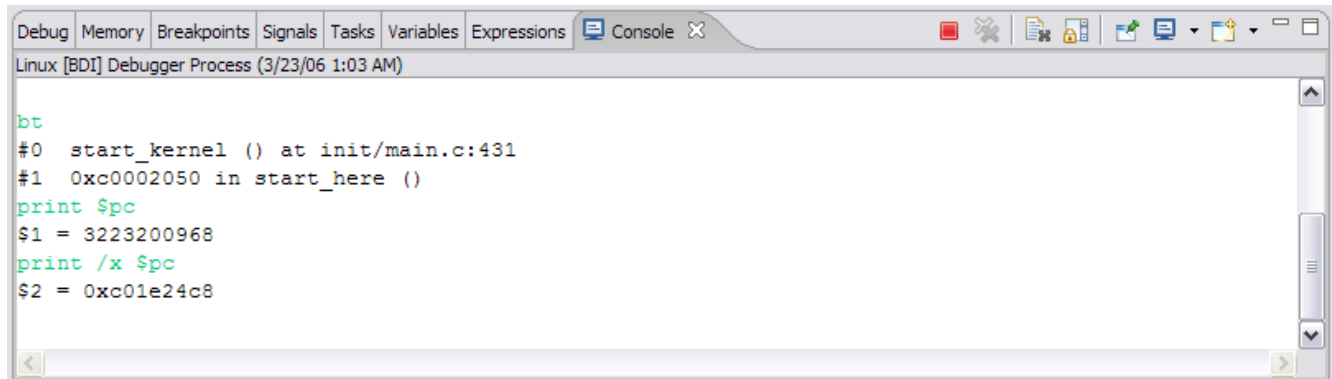
Double-clicking a stack frame opens the associated source file

The tool bar buttons, listed from left to right, are:

- 1.Remove Terminated Launches: Cleans up terminated debug sessions
- 2.Restart: Restarts the debug session
- 3.Load: Loads the binary image into the target via the JTAG probe
- 4.Instruction Stepping Mode: Step instructions rather than source lines; this must be selected if there are no sources or if the stack is not valid.
- 5.Step Into: Step into function or branch instruction
- 6.Step Over: Step over statement or instruction
- 7.Step Out: Continue execution until function exits
- 8.Run/Resume: Resumes target execution
- 9.Suspend: Suspends target execution
- 10.Connect: Connect to remote target; this is used if the session was launched without the “Attach to JTAG target on startup” option
- 11.Disconnect: Disconnects LinuxScope from the JTAG Probe but does not kill the debug session
- 12.Reset: Resets the target connected to the JTAG probe via the GDB “load” command
- 13.Sync: Synchronizes LinuxScope, GDB, and the JTAG probe; useful if the probe was accessed directly via Telnet

There are also Buttons in the tool bar which come from other Eclipse applications.

5.5 Console View



The screenshot shows the GDB Console window with the following content:

```
Linux [BDI] Debugger Process (3/23/06 1:03 AM)

bt
#0  start_kernel () at init/main.c:431
#1  0xc0002050 in start_here ()
print $pc
$1 = 3223200968
print /x $pc
$2 = 0xc01e24c8
```

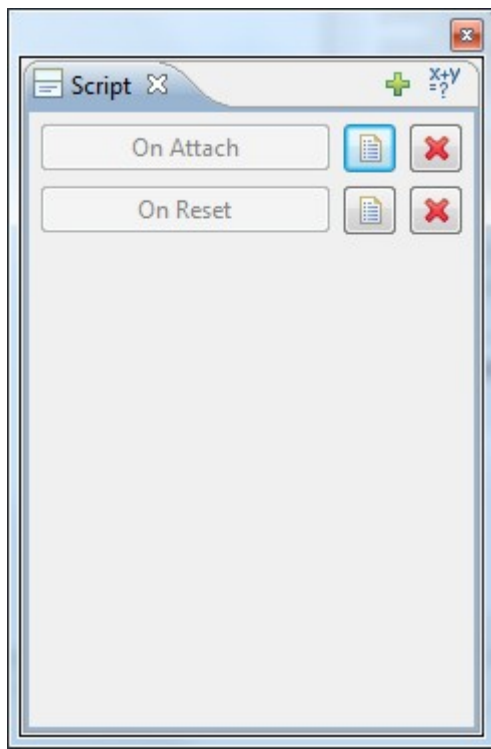
Direct interaction with GDB is possible by entering commands into the console for the GDB process. Black lines come from the output stream, red lines come from the error stream.

Mon Commands

Commands that are interpreted by the remote target rather than GDB can be sent with the “monitor” or “mon” GDB command. The set of “mon” commands available are the same as the set of commands accepted over Telnet connections to the JTAG probe. Entering “mon help” into the GDB console results in a list of these commands.

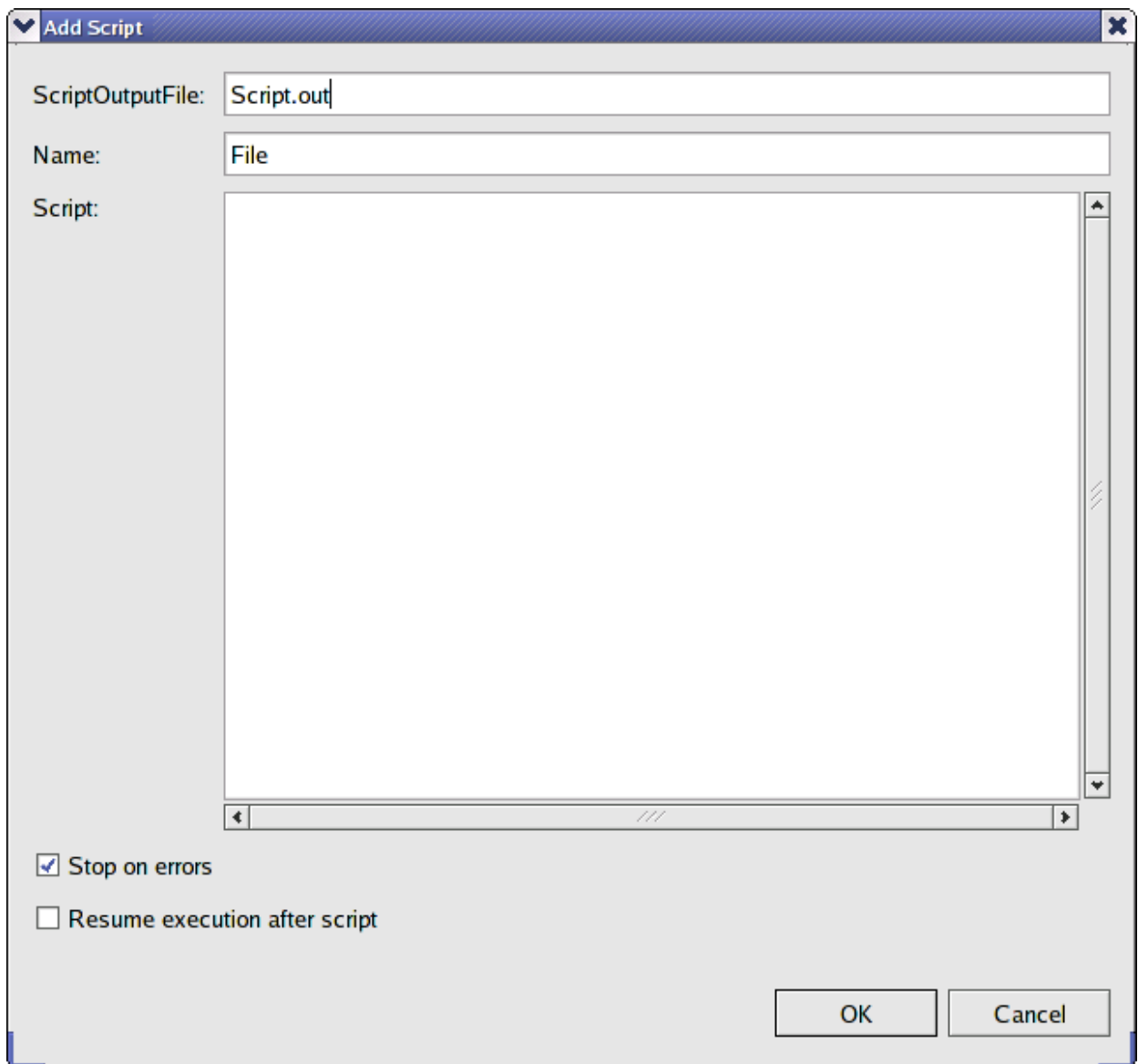
5.6 Scripting

The Script View provides a set of buttons that send commands to GDB or to the remote target.



Each script gets a button with a label; when pressed, this button runs the script. To the right are 'modify' and 'delete' buttons. The green '+' button is used to launch the script add/edit dialog and the blue 'x+y' button is used to launch a trace script dialog.

The script add/edit dialog contains several fields:



The screenshot shows a standard Windows-style dialog box titled "Add Script". It features a close button (X) in the top right corner. The dialog contains three main input fields: "ScriptOutputFile:" with the text "Script.out", "Name:" with the text "File", and "Script:" which is a large, empty text area with a vertical scrollbar on the right. Below these fields are two checkboxes: "Stop on errors" (checked) and "Resume execution after script" (unchecked). At the bottom right, there are two buttons: "OK" and "Cancel".

ScriptOutputFile: Filename to optionally capture the script's output. Script output directory is defined in the "JTAG Debug" Preferences. The script will not output to a file if this field is left blank.

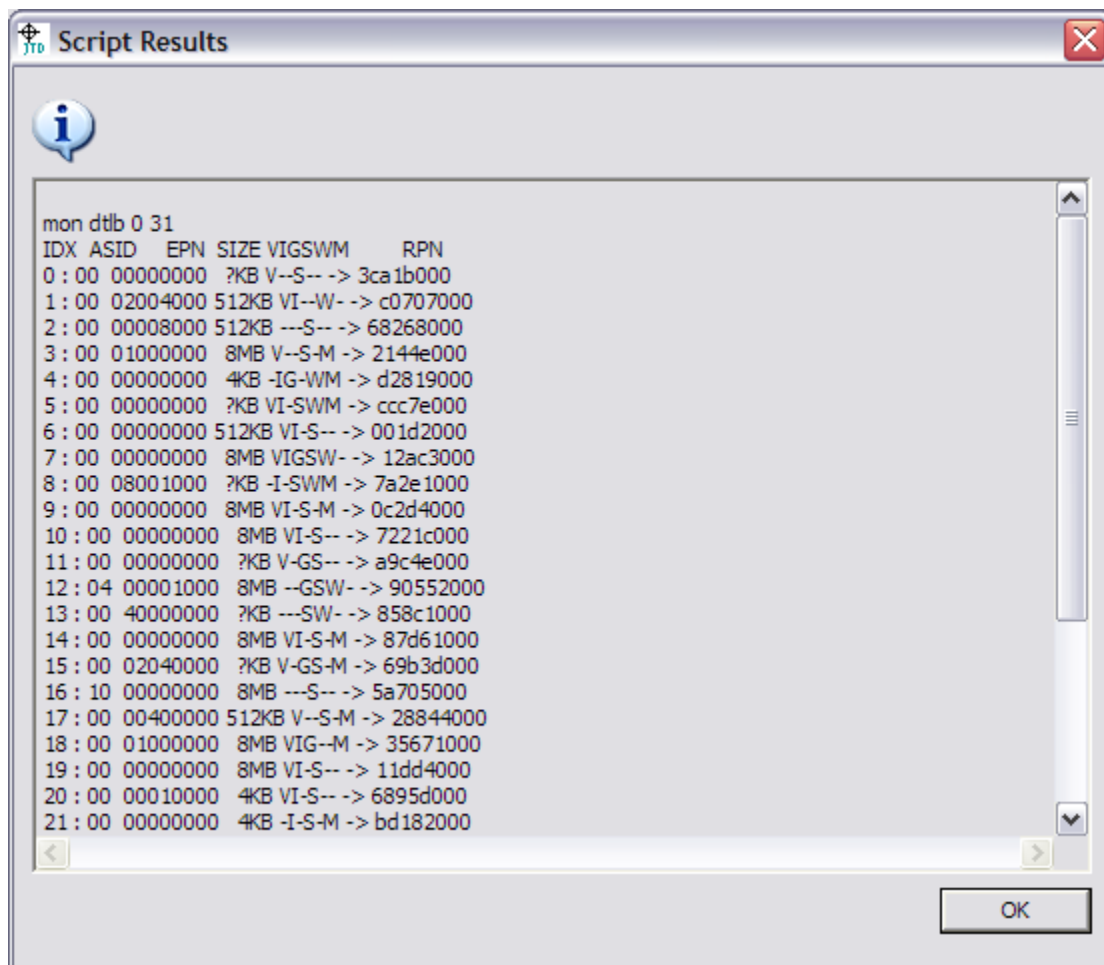
Name: Graphical display name; this need not be unique.

Script: List of commands to send to GDB. Lines starting with '#' are ignored. Lines ending with '\' are continued on the next line.

Stop on errors: If this is checked, the script will stop after any line that results in an error. Otherwise, execution will proceed.

Resume execution after script: If this is checked, the target will be resumed once the script is complete. Otherwise, it will remain suspended.

After the script runs, the output dialog appears:



The trace script add/edit dialog contains several fields:

Add a Trace Script

TraceScriptOutputFile:

Name:

Trace Location: Trace Loop Count (0=forever):

☐ Trace Stop Condition:

Check data to Store below:

☒ registers ☒ Variables ☒ function arguments ☒ local stack

Script:

☒ Stop on errors

TraceScriptOutputFile: Mandatory filename to capture the script's output. Script output directory is defined in the "JTAG Debug" Preferences. The trace script will not run if this field is left blank.

Name: Graphical display name; this need not be unique.

Trace Location: address or symbol name of breakpoint where trace will be captured.

Trace Loop Count: The number of times the breakpoint should be hit. Note that if this number can be very large but only the last 500 data points will be stored.

Trace Stop Condition: When this checkbox is enabled you can evaluate a simple expression as a condition to stop the trace experiment on. All the variables in the expression are calculated sequentially.

registers: When this button is checked the BDI will store all register contents at each trace iteration.

variables: When this button is checked the BDI will store local variables to the current function is any.

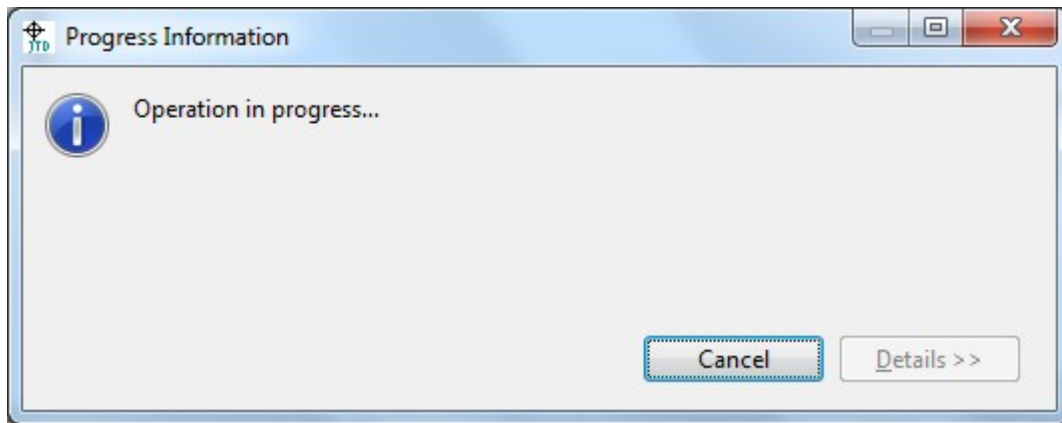
Function arguments: When this button is checked the BDI will store all arguments to the current function is any.

Local stack: When this button is checked the BDI will store the local stack frame.

Script: List of commands to send to GDB. Lines starting with '#' are ignored. Lines ending with '\' are continued on the next line. The script is executed every time the breakpoint is hit and the results are saved.

Stop on errors: If this is checked, the script will stop after any line that results in an error. Otherwise, execution will proceed.

After the script runs, the Progress dialog appears:



You can manually stop the trace script at anytime by clicking the cancel button. After the trace script finishes running it will process the output and save It into the specified file and directory.

Mon Commands

As with console views, scripts accept probe telnet commands if they are prefixed with the GDB “mon” command.

Special Commands

The script interface supports special commands that start with ':'. There are currently two special commands:

- :wait <milliseconds> - causes the script to pause before continuing to the next command
- :sync – causes Eclipse to sync with the remote target and refresh all the views
- :loop **n** – causes the script to be executed **n** iterations
- :endloop – terminates the sequence of commands to iterate

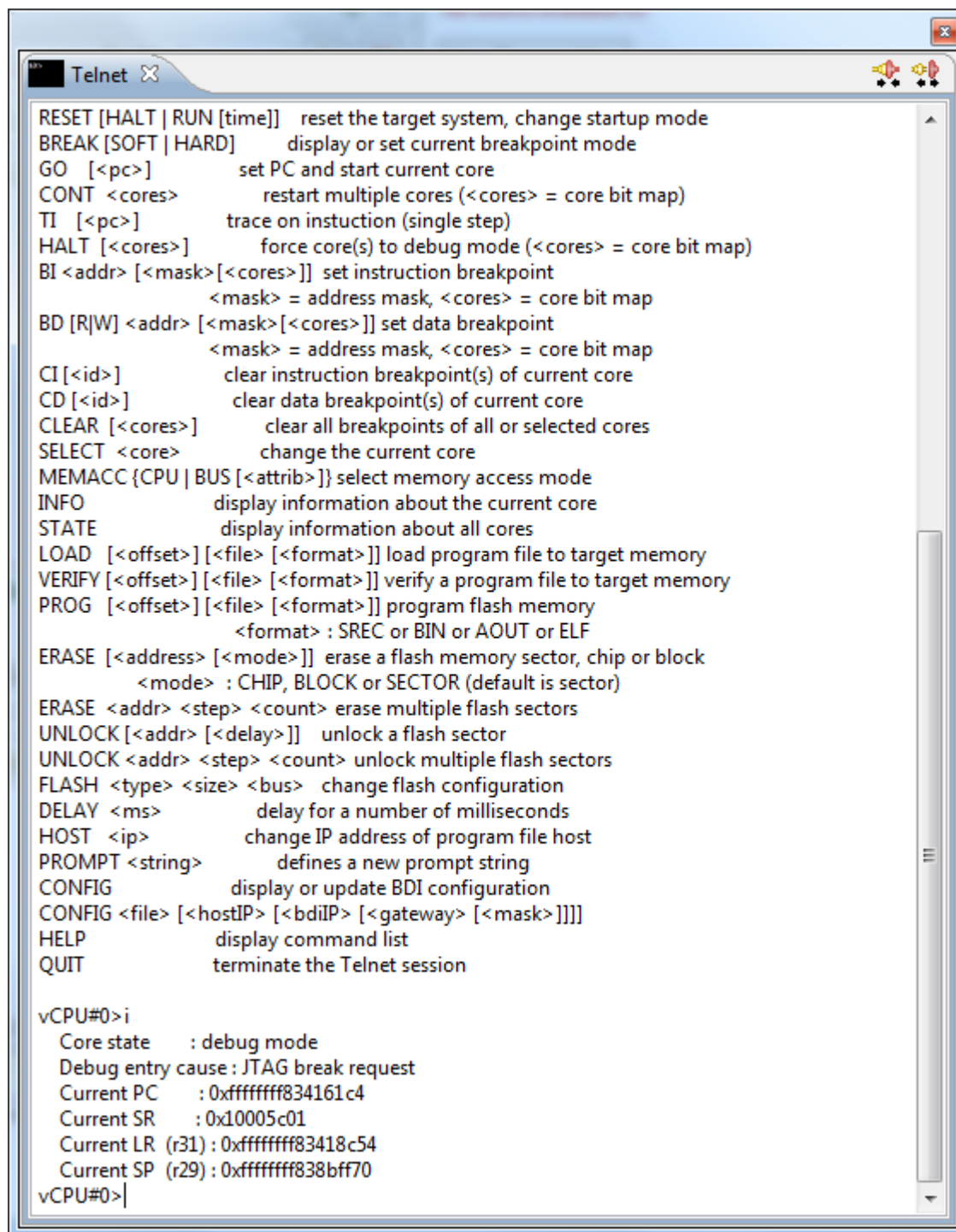
Automatically Triggered Scripts

The script view provides placeholders for scripts that are automatically run when certain events occur:

On Attach: Runs after LinuxScope connects to the remote target

On Reset: Runs after the target is reset (either via the launch process, or the reset button)

5.7 Telnet View

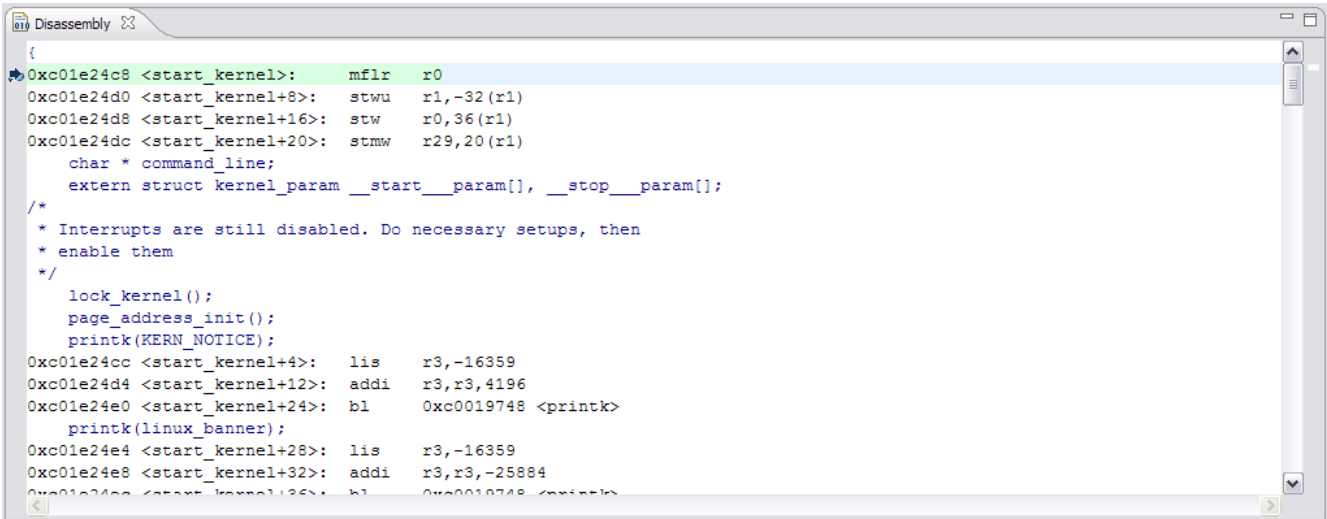


```
Telnet X
RESET [HALT | RUN [time]]  reset the target system, change startup mode
BREAK [SOFT | HARD]       display or set current breakpoint mode
GO [<pc>]                  set PC and start current core
CONT [<cores>]             restart multiple cores (<cores> = core bit map)
TI [<pc>]                  trace on instruction (single step)
HALT [<cores>]            force core(s) to debug mode (<cores> = core bit map)
BI [<addr> [<mask> [<cores>]] set instruction breakpoint
                           <mask> = address mask, <cores> = core bit map
BD [R|W] [<addr> [<mask> [<cores>]] set data breakpoint
                           <mask> = address mask, <cores> = core bit map
CI [<id>]                  clear instruction breakpoint(s) of current core
CD [<id>]                  clear data breakpoint(s) of current core
CLEAR [<cores>]           clear all breakpoints of all or selected cores
SELECT <core>              change the current core
MEMACC {CPU | BUS [<attrib>]} select memory access mode
INFO                       display information about the current core
STATE                      display information about all cores
LOAD [<offset>] [<file> [<format>]] load program file to target memory
VERIFY [<offset>] [<file> [<format>]] verify a program file to target memory
PROG [<offset>] [<file> [<format>]] program flash memory
                           <format> : SREC or BIN or AOUT or ELF
ERASE [<address> [<mode>]] erase a flash memory sector, chip or block
                           <mode> : CHIP, BLOCK or SECTOR (default is sector)
ERASE <addr> <step> <count> erase multiple flash sectors
UNLOCK [<addr> [<delay>]] unlock a flash sector
UNLOCK <addr> <step> <count> unlock multiple flash sectors
FLASH <type> <size> <bus> change flash configuration
DELAY <ms>                 delay for a number of milliseconds
HOST <ip>                   change IP address of program file host
PROMPT <string>             defines a new prompt string
CONFIG                     display or update BDI configuration
CONFIG <file> [<hostIP> [<bdiIP> [<gateway> [<mask> ]]]]
HELP                       display command list
QUIT                       terminate the Telnet session

vCPU#0>i
Core state      : debug mode
Debug entry cause : JTAG break request
Current PC      : 0xffffffff834161c4
Current SR      : 0x10005c01
Current LR (r31) : 0xffffffff83418c54
Current SP (r29) : 0xffffffff838b7f70
vCPU#0>
```

The Telnet view allows you to connect to a telnet session on the BDI. It uses the IP address of the selected debug configuration from the debug view. To connect to a session select the 'connect' button, to disconnect select the 'disconnect' button.

5.8 Disassembly View

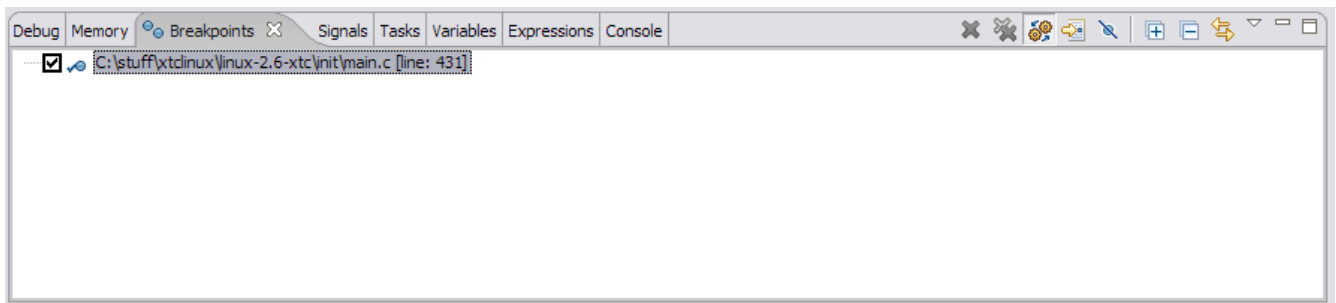


The screenshot shows a disassembler window titled "Disassembly". The code is for a function starting at address 0xc01e24c8. The assembly instructions are as follows:

```
{
0xc01e24c8 <start_kernel>:  mflr    r0
0xc01e24d0 <start_kernel+8>:  stwu    r1,-32(r1)
0xc01e24d8 <start_kernel+16>: stw     r0,36(r1)
0xc01e24dc <start_kernel+20>: stmw    r29,20(r1)
    char * command_line;
    extern struct kernel_param __start__param[], __stop__param[];
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
    lock_kernel();
    page_address_init();
    printk(KERN_NOTICE);
0xc01e24cc <start_kernel+4>:  lis     r3,-16359
0xc01e24d4 <start_kernel+12>:  addi    r3,r3,4196
0xc01e24e0 <start_kernel+24>:  bl      0xc0019748 <printk>
    printk(linux_banner);
0xc01e24e4 <start_kernel+28>:  lis     r3,-16359
0xc01e24e8 <start_kernel+32>:  addi    r3,r3,-25884
0xc01e24ec <start_kernel+36>:  bl      0xc0019748 <printk>
```

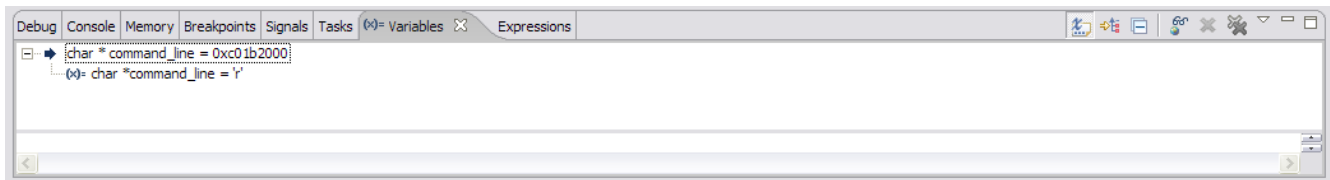
The disassembly view shows a mixed source and assembly view of the code referenced by the current stack frame. If no sources are found by GDB, this view will show assembly code only. The left ruler shows the current instruction pointer, and allows breakpoints to be set.

5.9 Breakpoints View

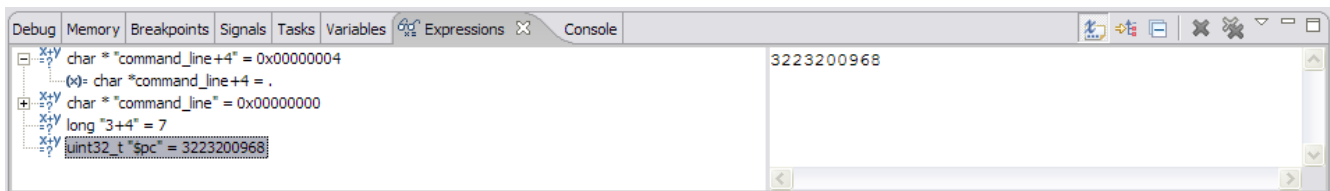


The breakpoints view is a simple list of defined breakpoints. It can be used to determine which breakpoints are loaded, and to toggle, enable, or disable breakpoints.

5.10 Variables, Expressions, and Memory Views

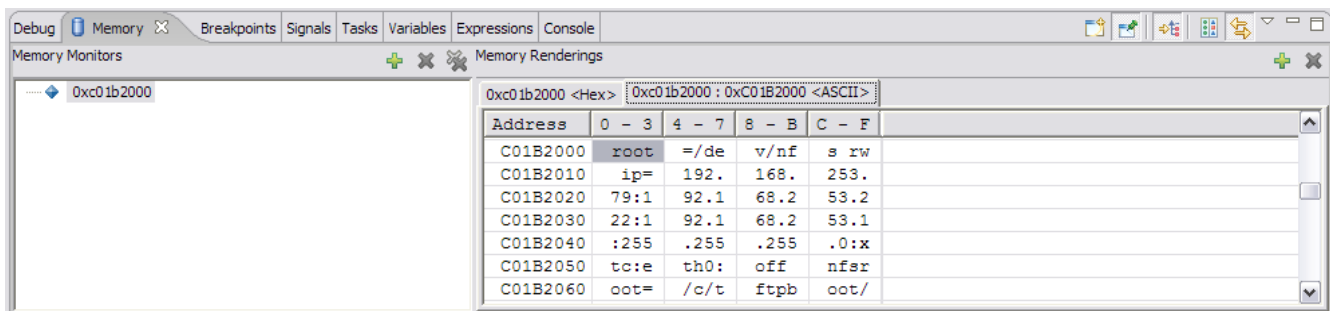


The variables view shows the values of local variables in the current stack frame. Global variables can also be added to the display.




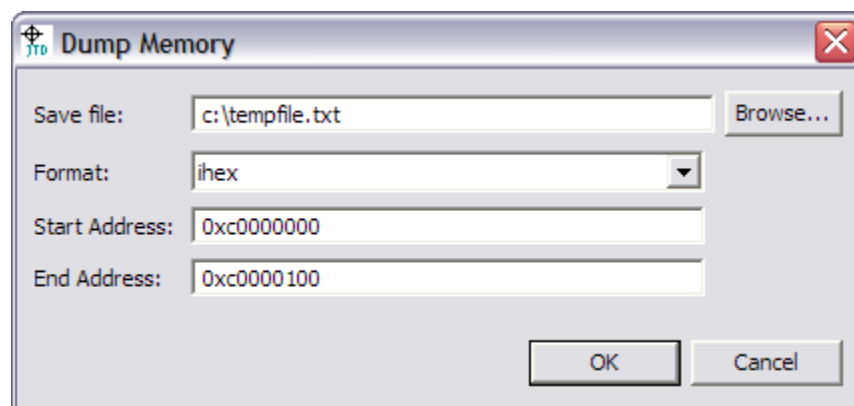
The expressions view can be used to enter watch expressions. Right-click to add expressions.

Note: Do not put newlines in expressions; expressions with newlines cannot be evaluated.



The memory view is used to display memory segments. Use the green plus icons to add memory regions and rendering formats to the display.

 The disk icon can be used to dump a memory region to disk.



6 Flashing

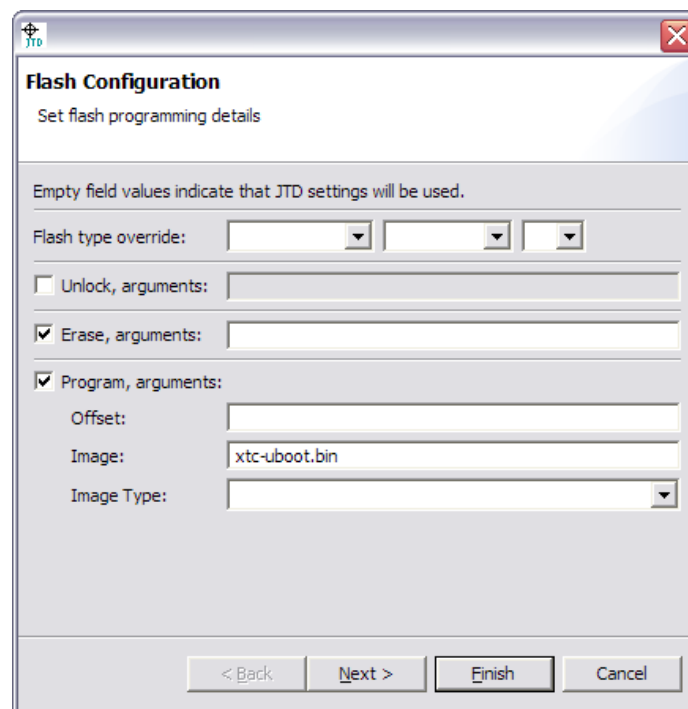
6.1 NOR

Select “Flash...” from the JTD menu or press the “Flash” toolbar button.



Note: Flashing creates its own target connection, so any debugging session using the JTAG probe must be ended before flashing. Most targets need to be reset before flashing; LinuxScope does this automatically.

The “Flash” wizard appears:

A screenshot of the 'Flash Configuration' dialog box. The title bar says 'JTD'. The main title is 'Flash Configuration' with a subtitle 'Set flash programming details'. Below this is a note: 'Empty field values indicate that JTD settings will be used.' The dialog contains several fields: 'Flash type override:' with three empty dropdown menus; 'Unlock, arguments:' with an unchecked checkbox and an empty text field; 'Erase, arguments:' with a checked checkbox and an empty text field; 'Program, arguments:' with a checked checkbox, an 'Offset:' text field, an 'Image:' text field containing 'xtc-uboot.bin', and an 'Image Type:' dropdown menu. At the bottom are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Select flash commands and arguments. If the arguments are empty, values from the BDI config file will be used. The image name is relative to the TFTP server.

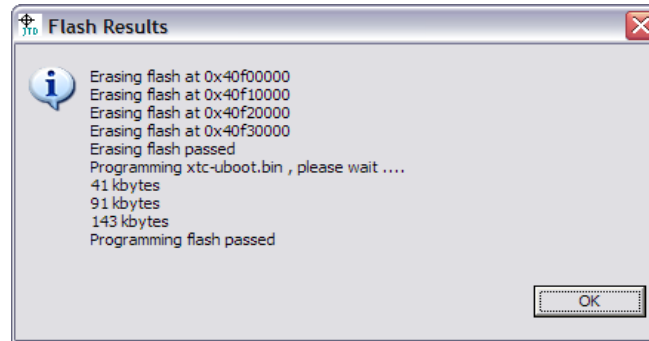
Flash type override: Use these drop down boxes to select your Flash configuration. Overrides the default arguments in the BDI config [FLASH] section file if specified.

Unlock, arguments: When checked it will run an “unlock” command to the BDI before programming. The arguments can be typed in as they would in the BDI Telnet interface. Overrides the default arguments in the BDI config [FLASH] section file if specified.

The arguments can be typed in as they would in the BDI Telnet interface. Overrides the default arguments in the BDI config [FLASH] section file if specified.

Program, arguments: Overrides the default arguments in the BDI config [FLASH] section file if specified.

The second screen of the wizard contains target settings; this step can be ignored if the global preferences are correct. Press “finish” to flash the target. Results are displayed:



Note: More sophisticated flash sequences can be accommodated with the Script interface.

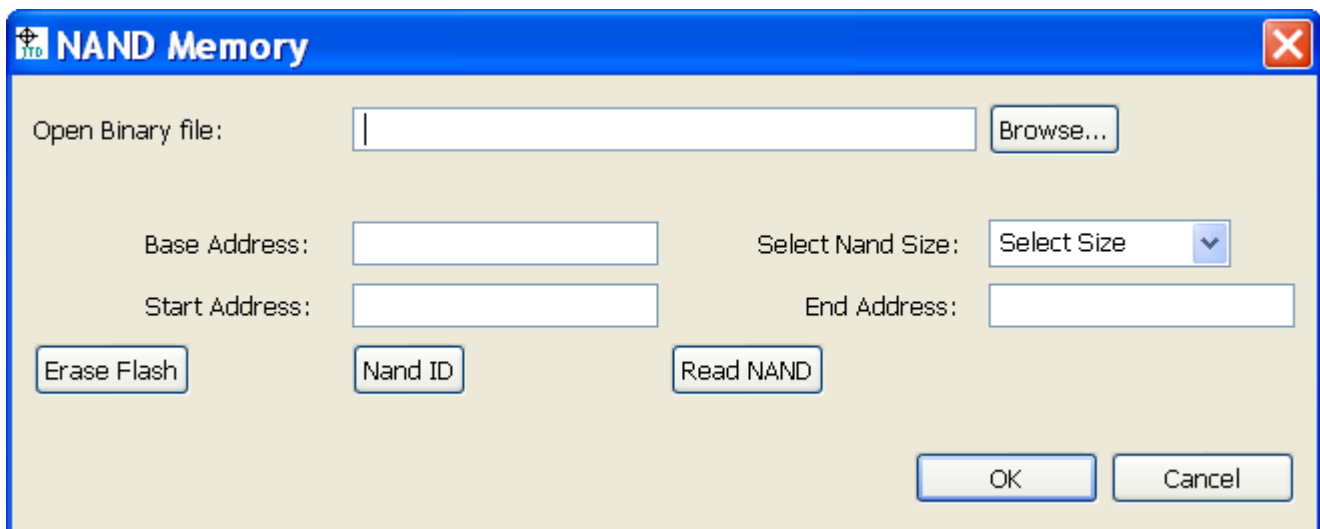
6.2 NAND

Select “Nand...” from the JTD menu or press the “Nand” toolbar button.



Note: Nand Flashing needs an active connection to the target. It will however initialize the target Chip Selects based on the parameters provided. Currently only the E500 Nand Controller is supported with a Large Page Nand Device on CS0.

The “Nand” wizard appears:



Open Binary file: Use text box to specify the file to burn into NAND.

Base Address: Specifies the desired base address location of the Nand Flash.

Select Nand Size: Size of the Nand device being used.

Start Address: Address where the binary file should be loaded in Nand, must fall in the Nand Address window specified by the Nand base address + Nand Size parameters. Also the start address for a Nand Read command.

End Address: The end address where the binary file should be loaded in Nand, must fall in the Nand Address window specified by the Nand base address + Nand Size parameters. If the file size exceeds Start Address -End Address then the file is clipped to fit the specified range. Also the end address for a “Read Nand” command.

Erase Flash: Erases the entire nand flash device.

Nand ID: Returns the Manufacture ID of the Nand Device..

Read Nand: Reads the Nand device from the range specified from Start Address till End Address..

7 General Tips

Debugging Boot ROMs and Other Bare Metal Programs

Use of hard breakpoints is recommended. Code can be loaded into the target using the load option/command for memory-resident programs once the target is initialized. Flash-based code can be loaded with the flash wizard.

Debugging the Linux Kernel

Use of soft breakpoints is recommended, because there are far more breakpoints available. However, the soft breakpoints cannot be set until the Linux MMU is enabled. This means that the first breakpoint hit must be a hard breakpoint, after which the soft breakpoints can be installed. Here is a simple set of steps to accomplish this:

1. Launch a debug session without attaching to the target, using soft breakpoints
2. Use the JTD Modules View search feature to locate the “start_kernel” symbol
3. Use the “Add Hard Breakpoint” action to set a breakpoint
4. Attach to the target, reset, and resume, so that the boot ROM can load the kernel
5. Once start_kernel is reached, soft breakpoints will be working, and all soft breakpoints will be automatically sent to the target on the next step/resume action

As long as the hard breakpoint remains set, this process does not need to be repeated. The hard breakpoint is saved with the project.

Debugging Linux 2.4 Kernel Modules

1. Use “insmod -m” when adding the module to the kernel
2. Paste the result into the “Add Symbol File” dialog when adding the module symbols

Note that in general, module initialization code can't be debugged in this manner because it runs before the symbol map is returned. If it crashes, the map can't be captured. In this case, the module must be rewritten so that initialization is deferred until after loading.

Debugging Linux 2.6 Kernel Modules

Because the module loading process has moved from user space into kernel space, insmod does not print out a map of what it did. However, if CONFIG_KALLSYMS is set, the kernel exports the map to the /sys file system, where it can be retrieved using a script such as the one in the section entitled “Adding Symbol Files”.

Application Debugging

Eclipse and CDT are useful for debugging Linux applications. (The hardware probe is not used for this.) Here are some tips:

1. Use the “Debug” perspective (not the “JTAG Debug” perspective)
 2. Run the application under “gdbserver” on the target
 3. Use the “C/C++ Local Application” launch type in the launch configuration dialog
 4. Use the “GDB Server” debugger on the debugger tab
- Use TCP for the connection, and specify the target's IP address